Tool Coordination in Software Development Workspaces

by

Nicholas Bradley

B.Sc., Queen's University, 2013
M.Sc., The University of British Columbia, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2024

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Tool Coordination in Software Development Workspaces

submitted by **Nicholas Bradley** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Computer Science**.

Examining Committee:

Reid Holmes, Professor, Computer Science, University of British Columbia
Co-Supervisor

Thomas Fritz, Associate Professor, Department of Informatics, University of Zurich
Co-supervisor

Gail Murphy, Professor, Computer Science, University of British Columbia
Supervisory Committee Member

Elisa Baniassad, Professor of Teaching, Computer Science, University of British Columbia
University Examiner

Ali Mesbah, Professor, Electrical and Computer Engineering, University of British Columbia
University Examiner

Thomas LaToza, Associate Professor, Computer Science, George Mason University
External Examiner

Additional Supervisory Committee Members:

Joanna McGrenere, Professor, Computer Science, University of British Columbia
Supervisory Committee Member

Abstract

Building and evolving modern software systems requires developers to use multiple tools within their workspaces. Unfortunately, the design of current workspaces, where tools operate independently of each other, leaves the manual and laborious orchestrating of these independent tools to developers. The constant tool coordination, navigation, and configuration represents unnecessary work that acts as a form of friction that impedes developer productivity. In particular, the workspace's tool-centric design silos the information developers need for their tasks. Due to the diversity of projects and tasks developers work on, even advanced tools like the IDE will never be able to solve this unnecessary work for all developers. Instead, new approaches are needed to overcome challenges developer's encounter locating and aligning information between their tools.

In this thesis, we examine the friction that developers experience from the tool-centric design of current workspaces, and explore two context-centric approaches, Helm and Scout, which use contextual information across tools to help mitigate the friction developers encounter when locating and aligning information necessary for their tasks.

Our first approach, Helm, is a lightweight system that automatically captures contextual information about the developer's project and the resources they access to reduce the effort of manually re-finding resources. Helm uses this context to recommend resources, including source code files, web pages, and commands, from a central location enabling developers to more directly navigate between resources. Our second approach, Scout, uses contextual information about the developer's source code to help the developer locate API information on the web. Scout tailors search results to the developer's task by extracting, ranking, and presenting the most relevant API signature information directly within the IDE where the information is needed.

Through two controlled studies, one with 17 developers using Helm and another with 40 developers using Scout, we analyzed how the approaches affected the way developers located information in their workspace. We found that, by using contextual information across tools, approaches can significantly help developers locate information for their tasks. Building on

these findings, we establish a new vision for the modern workspace where tools share contextual information to proactively support developers' tasks and improve productivity by reducing unnecessary friction.

Lay Summary

Software development is a highly information-intensive field where developers frequently switch between various tools to write code, manage tasks, and release software. This constant switching creates additional work as developers must locate and transfer information across independent tools which can significantly impact their productivity.

In this thesis, we identify the specific ways this extra work manifests and explore methods to reduce the burden of tool-switching and make existing tools more responsive to developers' needs. By sharing information about the files, web pages, and commands developers use across their tools, these methods ultimately enable developers to concentrate on the most critical aspects of their tasks.

Preface

This thesis includes work from five research articles:

1. Nick C. Bradley, Thomas Fritz, Reid Holmes. Sources of Software Development Workflow Friction. *Empirical Software Engineering Journal (EMSE)*, 27(7), 34 pages, 2022.

2. Nick C. Bradley, Thomas Fritz, Reid Holmes. Lightweight Interactive Resource Recommendation. 5 pages. In revision.

3. Nick C. Bradley, Thomas Fritz, Reid Holmes. Supporting Web-based API Searches in the IDE Using Signatures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 12 pages, 2024.

4. Nick C. Bradley, Thomas Fritz, Reid Holmes. Bridging Siloed Development Tools. 5 pages. In revision.

5. Nick C. Bradley, Thomas Fritz, Reid Holmes. Context-Aware Conversational Developer Assistants. In *Proceedings of the International Conference of Software Engineering (ICSE)*, 11 pages, 2018.

All five papers have been submitted to peer-reviewed conferences and journals. The first four papers correspond with the the thesis chapters starting from Chapter 3. Parts of the fifth paper, which was also the basis of my M.Sc., are included in Chapter 6 to concretize the discussion.

All five papers were co-authored with my research supervisors. I developed the approaches, conducted the studies, and analyzed the results, which form the core of this thesis. The studies were conducted under the UBC Behavioural Ethics Board certificates H17-02682, H21-03310, and H17-01251. I received guidance from my research supervisors. I drafted initial versions of the papers and made revisions based on peer-review feedback, with the assistance and editing of my supervisors. These papers appear mostly as they are written for publication, with some changes to organization and notation for consistency throughout the thesis.

The study instruments, tools, data, and analysis scripts that support this thesis are listed in Appendix A.

Table of Contents

Appendix

List of Tables

List of Figures

Acknowledgements

First, I would like to thank my co-supervisors, Reid Holmes and Thomas Fritz. This thesis would not have materialized without your dedicated guidance, unwavering support, and belief in this work. Reid, being able to drop by your office to discuss the next paper revision, 310 lecture, or biking trip was always extremely helpful to keep me grounded and focused on the next step. You've been a huge source of support and your almost immediate feedback always kept me going. Thomas, thank you for keeping me organized and focused on the end goal through these years. Your insightful comments and difficult questions made this work so much stronger; I'm still fond of our discussions at Dagstuhl and my two months in Zurich.

I would also like to thank my supervisory committee members Gail Murphy and Joanna McGrenere for their guidance and insight. I'd also like to thank other faculty and staff at UBC who have provided me with valuable support and assistance in completing my studies and working towards my career goals: Ivan Beschastnikh, Elisa Baniassad, Karen Flood, and Rachel Pottinger, to name a few. I'd also like to thank the CS department's administrative staff, help desk, and course coordinators, who have helped me in a variety of ways.

I've had the pleasure of interacting with many great students at UBC. Thanks to everyone in the SPL, and in particular, Felix Grund, Katherine Kerr, Giovanni Viviani, Arthur Marques, Braxton Hall, and James Yoo for the support and camaraderie.

Financial support for this thesis was provided by a Canada Graduate Scholarship from the Natural Sciences and Engineering Research Council of Canada and a Four-Year Fellowship from the University of British Columbia.

Finally, thanks to my family and friends for all their support. My wife Susanne has talked me off the proverbial ledge more times than I care to admit, while still providing more love and support than I could have hoped for. To my children Virgil and Agnes: thank you for keeping the Ph.D. in perspective.

To Virgil and Agnes.

Chapter 1

Introduction

Developers, like other knowledge workers, conduct the majority of their work digitally within their individually configured desktop workspaces. Using applications such as their integrated development environments (IDEs), command line interfaces (CLIs), and web browsers, developers access and manipulate information across different kinds of resources including source code files, bug reports, Q/A forums, and API documentation [12]. For any given task, developers use an average of four to five tools across these applications [81]. While IDEs integrate tools supporting the code-centric aspects of developer's tasks like navigation, debugging, and refactoring [8, 94], developers also spend a significant amount of their time, up to 40%, outside of the IDE [8, 12, 91]. The minimal support provided by IDEs to incorporate other important tools such as team collaboration, task management, and version control results in usability issues that limits developers' use of these integrated tools [21, 65, 94].

The complexity of modern development tasks requires developers to coordinate multiple different tools that rarely align directly with the task [74, 81, 89, 132]. To coordinate these tools, developers often have to breakdown their high-level tasks (e.g., sharing code changes) into sequences of low-level actions (e.g., switching to the CLI, navigating the project directory, and running the appropriate version control commands) representing their ad hoc workflows [136]. However, as most tools are independent, developers are forced to manually align each tool with their task, for instance, by navigating to the same resource in multiple tools (e.g., a source file in a code review tool and the same file in the IDE for deeper inspection), or by copying information between multiple applications (e.g., between an online issue tracker and the CLI while committing a change) [33, 81, 132].

Fred Brooks introduced the idea of accidental complexity to delineate the work essential to a task from the unnecessary work that acts as friction impeding task completion [27]. The work developers need to perform to align, coordinate, and configure the breadth of tools required to complete their tasks represents a meaningful source of unnecessary work [91]. Given that no single tool will work for all developers and all tasks, research has focused

on addressing this unnecessary work through automation. For example, Rich and Waters explored the idea of a programmer apprentice, which understands and applies standard approaches used by developers to automate parts of their workflows [117]. Early approaches used planning techniques to automate workflows performed on the command line (e.g., [20, 57]). While these early efforts were not feasible when they were proposed, recent advances in AI/ML have renewed interest in these approaches [48, 131]. Unfortunately, even these modern tools, while performing valuable work, still require developers to perform unnecessary work to coordinate them within their existing workspace.

Although LLM-based assistants like CoPilot [1] have shown great promise in automating some parts of development tasks, developers will still be required to coordinate this new class of tools within their workspace [29], giving rise to more unnecessary work as developers integrate these tools with their traditional development processes. In prior development tool research, the notion of context has been widely used, although typically within individual tools, to try to gain some insight into developer needs, for instance by organizing information around methods and their relationship to other resources [96]. This context can also improve developer productivity by helping developers focus on task-relevant content [66]. Ultimately, the diversity of tools developers need to coordinate, along with the added complexity induced by frequent task switching [44, 89], requires further investigation into approaches supporting workspace-level coordination to enable both existing and new tools to better support developers' focus on the essential parts of their tasks without being impeded by unnecessary work.

One frequent reason developers need to coordinate their tools is to locate information for their tasks [74, 132]. As the resources developers navigate to are related by the context of the task [33], it should be possible to help developers locate information across tools based on the resources developers have previously accessed. Specifically, metadata about the resources, such as the location within the workspace's hierarchical organization structures (i.e., filesystem paths, URLs), may provide information about the relationships between resources across tools (e.g., resources that belong to the same project). Content within resources may also act as a source of context for developers' tasks. For example, the programming language and library names used in a coding task help developers scope the search results when searching for solutions [54], while type information helps developers identify specific solutions that fit within their existing code [84].

The thesis of this work is that the friction developers experience locating and aligning information within their workspace can be reduced by automatically collecting and using context from the resources developers access across tools.

A key underlying assumption of our thesis is that friction impacts developer productivity. Thus, we first examined the research question:

RQ1 What kinds of frictions do developers experience in their desktop workspaces?

To address this research question, we analyzed screen recordings of 27 developers, 17 working on a given task in a controlled setting and 10 working on their own tasks. Based on the analysis we found that developers routinely perform extra work, such as locating and integrating information across tools, which acts as friction limiting how easily developers can complete their tasks. The findings of the study illustrate how the lack of coordination between tools can make it difficult for developers to find and re-find information for their tasks.

In the second step of this thesis work, we explored approaches that address the friction associated with locating information in the workspace by capturing and using context between tools. In one approach, we gather and use information about the developer's project and locations of past resources across tools as context to support developers' general navigation between tools. In another approach, we go deeper into supporting a common case where developers frequently have to switch tools: to look up information on the web when coding. We use specific context about the developer's code to tailor the information to better support the developer's task. We examine these approaches with the following two research questions:

RQ2 How effective is a centralized resource-centric approach at helping developers re-find and directly navigate project resources across tools?

RQ3 Can source code be used to tailor content within web page resources to make relevant information easier for developers to locate?

In the following sections we describe how we identified sources of friction in developers' workspaces, and provide background about the approaches and how we evaluated them in relation to the thesis statement. We first provide a list of terms that we use throughout this thesis to clarify their meaning.

1.1 Definitions

These terms are based in-part on the task framework developed by [28]. We use this framework to disambiguate the various uses of task (e.g., [37, 102, 116]) and activity (e.g., [11, 14, 59]) found in the literature.

Workspace. In this thesis, the workspace is the desktop environment that runs on the developer's machine and provides a graphical user interface (e.g., the MacOS and Windows desktop environments).

Applications and tools. Applications are programs that run within the workspace and enable developers to view and modify their resources. Examples of applications commonly used by developers include the integrated development environment (IDE), command line interface (CLI), and web browser. Tools provide more specific functionality within an application, for example, the linter in the IDE. For the purposes of this thesis, we largely use the terms *application* and *tool* interchangeably.

Resource. Any object represented in the workspace that enables developers to record and communicate information to themselves and other developers. Common resources include files, documentation, and issues. We also consider shell commands and tool panes within applications to be resources as they communicate information to developers. For example, the file and search panes in IDEs provide information about the structure and content of projects while shell commands like `git status` provide information about changes to the project.

Action. Any operation a developer can perform on a resource.

Task. In this thesis, task refers to the description of the work to be done by the developer; e.g., a bug report or todo items.

Goal. The objective towards which a developer's effort is directed to complete a task [33]. Goals are specific to individual developers and reflect the multiple ways that a task can be completed.

Workflow. The sequence of actions that developers perform in pursuit of their goals and tasks.

Activity. A descriptive high-level label summarizing observed actions using one of several researcher-defined categories. Activities simply aggregate developer actions and do not depend on the developer's current task or working context. For example, the action of setting a breakpoint could be described as a debugging activity without considering the developer's overall task.

Friction. Extraneous actions and mental effort that arise when developers' goals cut across tools and resources in the workspace.

1.2 Identifying Friction in Developer Workspaces

Given a task description, a developer's job is to alter the software system in a way that accomplishes their task, usually by fixing a bug or adding a new feature. Completing these tasks typically requires developers to use multiple tools within their workspace. In Chapter 3, we investigate how developers use their tools to perform their tasks and the ways in which these tools create unnecessary work which acts a friction.

In particular, to address our first research question, we conducted two studies: a controlled lab study with 17 developers working on a realistic task on a medium size open-source project and a study of 10 developers working in a in vivo setting working on their own tasks. Taking inspiration from the cognitive walkthrough method [77], we examined participants' workflows through a goal-centric lens to identify discrepancies between the tasks they wanted to accomplish and how they were able to operationalize them as sequences of tool actions. Together, these two studies allowed us to investigate how developers decompose a diverse set of tasks into goals and the frictions they experience operationalizing these goals into low-level action workflows across tools.

Based on data collected from the controlled study, we found that developers often derive similar sequences of workflows when decomposing the same task and operationalizing these tasks as action workflows is an indirect process that requires developers to perform actions and move information across multiple tools and resources. While operationalizing their tasks, developers plan and adapt the low-level actions needed for their high-level tasks (translation friction), combine information across resources dispersed in the workspace (integration friction), and locate resources in different organizational structures (access friction).

A key finding of this work is that locating and integrating information dispersed across different tools and windows is one of the most common ways developers encounter friction in their workspaces. With the following two approaches, we examine whether we can mitigate this friction by sharing context across tools.

1.3 Coordinating Project Resources Across Applications

Developers constantly switch between applications and resources as they work through their many interleaved tasks. Each of these switches requires

the developer to open the right application and re-find the resource they need. Once open, the developer can then use the resource to make progress on their task. In Chapter 4, we introduce a prototype tool, called Helm, that provides a centralized interface where developers can re-find and directly interact with their resources. To support re-finding, Helm tracks the resources a developer interacts with across applications, in particular the web pages in the browser, files in the IDE, and commands in the CLI. For each resource, Helm computes a relevance based on recency and frequency metrics, and associates it with the active development project. When a developer wants to locate a resource, Helm determines the currently active project and provides the top ten relevant resources. Once identified, Helm loads the resource so the developer can inspect and interact with it directly. For CLI commands in particular, which are frequently needed by developers but difficult to combine for a task, Helm enables developers to re-construct and re-execute shell scripts from their past commands.

We evaluated Helm in a controlled user study with 17 developers working on bug-fixing tasks for two real software projects. During the study, we recorded the resources the developers used and simulated typical working conditions by having the developers switch tasks midway through the study. When resuming the original task at the end of the study, the developers used Helm to re-find the required resources and re-establish context. We found that Helm provided useful recommendations for 80% of the resources developers revisited and, based on interviews with participants, found that Helm supports the identification of relevant resources and saves time, with actions available directly within the interface.

1.4 Aligning Resources with Tasks to Aid Information Navigation

Developers frequently search for new information about their coding task using web-based search engines like Google [149]. One of the most common searches is to locate and obtain information about application programming interfaces (APIs) relevant to their current task [2, 52, 54, 79, 127, 149]. Unfortunately, finding API information introduces workflow friction as developers have to frequently switch between their IDE and browser [105, 106], copy code terms into their searches [79, 150], and manage a large number of browser tabs as they assess different solutions [32, 92]. Even when a web page, such as a Stack Overflow (SO) post, contains the required API information, developers have to read through the content to identify which APIs

are present and how they might fit into their code [56]. These extra steps affect developers' focus and increase their cognitive load [40].

To address some of these challenges, researchers have investigated approaches to help developers find information relevant to their development tasks. Early web-based code search approaches, such as Mica [138] and Assieme [52], augmented search results with links to code elements contained within the search result pages to help developers refine the results. With the growth of SO as an essential source of development knowledge, approaches have focused on summarizing SO posts to answer a developer's query. For example, AnswerBot uses natural language processing to combine the textual information from multiple posts to create diverse summaries without focusing specifically on providing API information [150]. Biker provides summary documentation for specific APIs relevant to a developer's search query but requires a custom search engine indexed on SO data dumps and the official Java documentation [56]. More recent tools based on large language models (LLMs), such as ChatGPT and Copilot, offer alternative ways for developers to obtain solutions to their tasks. However, the solutions produced by these approaches can be incorrect [100], verbose [62], and often require repeated refinement of the queries [98], making it difficult for developers to understand and compare solutions.

In Chapter 5, we introduce an approach, called Scout, which succinctly represents the results of a search query using API signatures to make it easier for developers to identify and compare information relevant to their API search tasks. Specifically, Scout adapts the presentation of SO Google search result pages so developers can more directly compare alternative solutions in the limited space of the IDE by hiding non-essential and duplicate information.

Scout extracts *signatures*, consisting of a method name, class type, parameter type(s), and return type, from the SO posts in the developer's search results. These signatures are presented as focus points, giving developers an overview of potential solutions without having to examine each post individually. This overview encourages top-down information gathering where developers first identify the most appropriate signatures before examining detailed code examples and descriptions from the source SO posts.

Scout is integrated into the IDE enabling developers to assess their search results directly from their code without losing focus on their current task. Using this integration, Scout also automatically derives terms, such as variable types and library names, from the code the developer is working on as context to improve the API signature recommendations. When making a search, Scout suggests some of these terms to help scope the developer's

query. Once the developer has performed their search, Scout extracts the embedded API call signatures from each SO post in the search results. Scout presents the top-three signatures under the title of each post after ranking the signatures by their frequency of occurrence and the number of types they share with the developer's context.

To evaluate Scout, we conducted a controlled experiment with 40 developers. In the experiment, we asked participants to complete six coding tasks, randomly assigning them to either our experimental treatment, which presented API call signatures, or a baseline treatment which presented regular Google search results within the IDE. For the study, we implemented Scout as a VSCode IDE extension that ran directly in participants' browsers, recording their interaction and feedback as they completed the study tasks.

Overall, Scout made the most relevant information more directly accessible, resulting in participants completing almost half of the tasks directly using Scout's signature presentation and significantly reducing the number of posts opened by participants by 64%. Further, participants valued Scout's API-centered presentation, and Scout's automatically inferred context terms helped to significantly reduce the number of searches that developers performed.

1.5 Contributions

This thesis makes several contributions in the space of tool coordination. We identify design aspects of current workspaces that create friction for developers completing their tasks and evaluate two context-centric approaches that demonstrate that sharing a few pieces of information across tools can help developers locate information more directly. Specifically, we provide:[1]

1. **A systematic investigation of friction developers encounter completing tasks in their workspaces**; including:
    - A dataset of workflows from 27 developers annotated with their goals and the challenges they encountered when trying to complete their goals.
    - An analysis of the ways developers decompose and complete their tasks across applications.
    - A set of seven kinds of friction from a thematic analysis of the challenges developers encountered in their workspaces.

[1]The datasets and analysis for each of these contributions are publicly available and listed in Appendix A.

2. **An evaluation of a prototype tool exploring whether a resource-centric approach helps developers locate information for their tasks**; including:

    - A proof-of-concept prototype demonstrating that resource-centric approaches can eliminate much of the redundant application- and window-based navigation necessary in current workspaces.
    - Empirical findings which establish that information about the resources developers access across applications and projects, can assist developers re-find resources and re-establish the context necessary to resume their tasks.
    - A dataset of application switches and resources used by 15 developers in a controlled study to address real bugs in two open-source projects.

3. **An evaluation of a context-driven approach to extract and present API information from web resources within the IDE**; including:

    - An interactive presentation, integrated into the limited space of the IDE, that orients search results around API signatures enabling developers to first identify possible solutions before going into details, if they are needed.
    - A ranking algorithm for identifying and de-duplicating candidate APIs in the search results by considering type information within the developer's source code where the API will be used.
    - A dataset and empirical findings from a controlled experiment with 40 developers demonstrating that our approach supports developers in finding the information they need to complete API tasks more effectively than traditional web search.

These contributions establish that developers encounter friction in current tool-centric workspaces, and that this friction makes it difficult for developers to locate and align information for their tasks. Based on our analysis of the data we collected from our two studies, we found that context-centric approaches, like Helm and Scout, enable developers to more directly locate the information they need for their tasks.

1.6 Organization

The direction of this thesis is based on our preliminary observations of developers' work where we establish the way developers use tools to complete their tasks across applications (Chapter 3). Our aim was to identify how the design of tools, in particular the ways they work together, help or hinder the developer's tasks. Identifying that tool independence created friction for developer navigation within and between their tools, we evaluated two models incorporating information shared between tools to improve navigation: Helm (Chapter 4) helps developers directly revisit resources avoiding the overhead of re-navigation, while Scout (Chapter 5) directs developers to key information for API searches. Finally, we synthesize our results in a discussion of future research directions towards realizing a workspace that can align tools by sharing information (Chapter 6).

Chapter 2

Related Work

Developers use a host of applications within their desktop workspaces to complete their tasks. Early applications were usually simple programs designed to emulate the functionality of existing physical devices (e.g., calculators, text editors, etc.). These applications were designed to operate independently from one another, like their physical counterparts, and this design is still the one predominately used by every major desktop workspace today [134]. This application-centric design made sense when desktop workspaces were first being developed: individual applications which were direct analogues of existing tools made the system easier for users to understand and use. Even today, application-centric designs have many benefits: the simplicity of the design provides flexibility in terms of how the application is designed and implemented, while the independence of the design helps ensure privacy and security by limiting programmatic access to content within applications. However, application-centric designs can also act as artificial boundaries which can silo information [144]. These boundaries make locating and accessing information across the multiple applications developers require to accomplish modern development tasks more difficult [44, 81].

One approach to overcome some of the limitations of independent applications is to create "bigger" applications that incorporate functionality of multiple applications that are commonly used together [64]. For example, integrated development environments (IDEs) originally incorporated three common applications, a text editor, compiler, and debugger, that developers iteratively switched between as they built their software system [7]. Instead of developers switching between these tools and manually linking the actions and feedback with the file names and line numbers in the source code, IDEs enabled developers to immediately see the source of compilation errors and directly mark locations in the source code where the debugger should pause. Unfortunately, it is impossible to incorporate all the functionality required by all developers for all tasks into a single application. As features have been added to IDEs their design limitations are becoming apparent: the breadth of IDE features impacts usability [21, 93], the panel-based design

can cause developers to become disorientated across document and tool windows [36], and space constraints limit the amount of focused content that is visible [130].

A systematic investigation into the relationships between developers' tasks and their tools is necessary to understand the impact of these limitations on developers' work and to identify effective solutions. Prior work has examined ways that tools could be improved based on how developers structure their work (e.g., to better support multi-tasking), spend their time (e.g., time spent outside the IDE), and use tools complete their tasks (e.g., to identify missing tool support). Given the information-intensive nature of software development, prior work has also focused on understanding and improving the ways developers access information for their tasks across their tools; in particular, by helping developers organize and integrate information, and by recommending information directly to developers as they are working.

We extend this prior work by identifying the frictions developers encounter while completing their cross-application workflows, and show that contextual information helps to mitigate these frictions.

2.1 Connections Between Tasks and Tools

Developers complete the majority of their work digitally. The design of the tools and the desktop workspace environment have a direct impact on developer productivity [90]. Aligning workspaces with the work developers need to perform can make it easier for developers to complete their tasks efficiently [61, 73]. By understanding how developers structure and perform their tasks, gaps between what developers need to accomplish and what is supported in the workspace can be identified and addressed.

2.1.1 How Developers Structure Their Work

A developer's productivity is affected by how their workspace supports their working style [90]. Externally, a developer's work is often structured into discrete work items or tasks which are tracked within the project. While working on these work items, sub-tasks often emerge as developers encounter unexpected obstacles to their original high-level task [96]. Roehm et al. referred to the sub-tasks arising from these obstacles as problem-solution cycles [33, 121]. In some cases, developers encounter obstacles which block their task as they wait for information from their tools (e.g., build status) or colleagues (e.g., the rationale of a particular design decision) [70]. In other

cases, developers may be interrupted to attend to a unrelated task (e.g., responding to a colleague's question) [70].

As a result of these interruptions and unexpected sub-tasks, developer's frequently multi-task, switching between tasks every two minutes, on average [44, 89, 106]. While personality affects a developer's propensity to multi-task, the structure of the project is also an important factor [86, 143]. Despite developer's frequent task switching, it is not well supported in workspaces causing long resumptions lags as developers manually realign the workspace for their task [85]. In particular, as developers use the same applications for multiple tasks, it can be challenging for developers to identify the set of resources relevant for their current task [11, 44].

Gonzalez proposed the idea of working spheres as a way for workspaces to help developers easily switch between their tasks [44]. However, the emergence of (sub)tasks as developers are working, as well as the interleaving of tasks and applications in the workspace, means that the idea of working spheres has not been fully realized [156]. Identifying the steps developers perform to align their workspace when they switch tasks, and the friction associated with these steps, could lead to improvements in tool design that would enable working spheres to be implemented. Given the frequency of developers' task switches, even small improvements that help developers locate and align resources with their (sub)tasks could accumulate to significantly improve developer productivity over the course of a work day.

2.1.2 Quantifying Tool Usage for Development Work

Understanding how developers spend their time using different tools in their workspaces can help developers gain insights into their productivity, for example, by reflecting on the time they spend in different applications or working with specific resources [89, 123]. Information about tool usage can also be used by tool designers and researchers to help developers identify and use existing features more productively. For example, Murphy et al. monitored the tools developers used in the Eclipse IDE and found that developers mostly navigated manually between code elements in the package explorer hierarchy [94]. Based on this finding, Viriyakattiyaporn created an active help system, which monitors the actions developers use to navigate their code, to help developers learn about more efficient navigation tools available in the IDE [145].

Another way to build an understanding of how developers use their tools is by identifying the time developers spend doing different activities (Table 2.1). Activities represent small discrete components of developers' tasks

which generalize across developers and projects. By instrumenting developers' applications to capture low-level interaction data, researchers have identified the amount of time developers spend on different activities, initially starting with the IDE (e.g., [8, 91]). Recognizing that developers spend nearly 40% of their time outside the IDE [8], Bao et al. developed an approach to capture interaction data automatically across any application which adheres to the workspace's accessibility standards [13]. However, as interaction data is not associated with what developers were trying to accomplish during their tasks, it can only be used to describe what developers do, and does not capture the challenges they encounter using their tools.

2.1.3 Identifying Missing Tool Support

Development tasks are inherently difficult due to the size and complexity of the software systems developers need to understand to solve those tasks. One way to help developers complete their tasks more efficiently is by providing tools that directly support what developers need to accomplish [73], prompting researchers to examine how developers use their tools in the context of their tasks. Unlike studies quantifying developer's tool usage, these studies apply qualitative methods to understand the developer's task and identify limitations of existing tools in relation to what the developer wanted to achieve.

Through interviews with 62 software professionals, Maalej identified eight problems developers encounter integrating information across the four to five tools developers use, on average, to complete their tasks [81]. LaToza et al. surveyed 344 developers at Microsoft to understand the relationships between nine predefined development activities, the tools developers use to complete these activities, and the mental models developers maintain as they switch between these activities and found that developers use a variety of tools and workflows that they actively work to improve [74]. Several researchers have also directly observed developers working with their tools. For example, after shadowing a developer over several months to identify common activities, Singer et al. identified a lack of search support from the company's tool usage logs [133]. More broadly, researchers have found that developers' tasks can be blocked if they are unable to obtain necessary information but that getting this information often requires developers to break down their needs into tool-oriented questions and reintegrate the individual, possibly incompatible answers using "awkward series of actions" [70, 132]. However, little is known about the way developers perform these actions and the challenges developers face when completing

their tasks across tools.

2.2 Locating Information Across Tools

Developers need larges amounts of information to accomplish their tasks [33, 72, 78], for example, important locations in source code [22], variable states in the debugger [71], and API documentation [56]. Navigating between tools and resources for this information can be tedious as developers are often guided only by estimates of where they believe the information is located [108]. To help reduce this costly navigation, approaches have broadly focused on organizing resources so developers are more likely to find the information, integrating information across resources into a single location, and recommending related resources to make navigation more direct.

2.2.1 Organizing Cross-Application Resources

Developers revisit their resources to resume an interrupted task or recall information such as the details of a bug report or a solution to a coding problem [44, 106, 129]. Several approaches allow developers to manually associate their resources with a task to help them find their resources more easily (e.g., [37, 59, 63, 66, 118]). However, developers need to remember to mark the start and end of their tasks for these approaches to be effective which can be difficult to remember and creates overhead for the developer [113].

Alternatively, approaches have attempted to group resources automatically by tracking metadata (e.g., window titles [101]) and developer interactions (e.g., [19, 116, 122, 140]). However, the tasks these approaches infer may not always align with the actual tasks developers are working on, especially considering developers frequent multi-tasking, making the resources difficult for developers to find.

2.2.2 Integrating Disparate Resources

Developers require information provided by specialized tools, e.g., debuggers, versioning tools, to complete their tasks [70]. Coordinating these tools is not always easy as the format of the information varies and is separate from where it is needed requiring developers to manually integrate the information [72, 81, 132].

Copy/Paste is one basic approach developers use to move information between resources [5], but requires developers to often navigate between two

or more resources repeatedly. By making assumptions about the information developers are likely to copy, approaches can reduce repeated navigation by making the information available where it will be pasted [139, 155]. For example, Augur autocompletes web-based form fields based on information from previous pages the developer visited [50].

Instead of manually interacting with different tools, approaches that provide centralized access to information can enable developers to express their information needs more directly. For example, CodeBook creates a centralized graph relating information stored across siloed databases, e.g., employee directories, version control systems, and work items, enabling developers to directly query for the required information from a single location [17]. Similarly, Fritz and Murphy introduce composition and projection operators over a graph associating information fragments to allow developers to answer seven broad types of questions by linking and restructuring information hierarchies [39].

The way information is presented to developers can also affect the difficulty of their tasks. By bringing together information from different tools and resources across the workspace, approaches can provide different perspectives to help developers better understand, and more directly complete, their tasks. For example, Codelets presents API code snippets developers copied from the web as views integrated directly in the code editor that developers can interact with to understand the behaviour of the API [103]. The Whyline displays the state of code elements during program execution that affect the behaviour of a particular feature to reduce developers' effort manually stepping through the code using a debugger [71]. Canvas-based IDEs present code files as views of individual methods which developers can arrange to understand program flow and cross-cutting features [23, 36, 51]. By incorporating annotations as an additional type of view within a canvas-based IDE, Adeli et al. showed that positioning relevant information close to where it is needed increases accuracy and reduces the cognitive load of new developers performing code comprehension tasks [3]. While these approaches support developers with specific kinds of information, developers still need to coordinate them with their overall task.

2.2.3 Recommending Task-Relevant Resources

Developers frequently access information in other resources to complete their tasks [106]. Rather than locating the necessary resources manually, approaches can help developers stay focused on their main task and improve their productivity by recommending resources automatically as developers

are working [119]. These approaches primarily differ in whether they recommend resources developers have previously accessed or new resources. For example, Hipikat recommends the discussions, bug reports, documentation, and file revisions developers have previously accessed as they are working on source code in their IDE [157]. Reverb recommends web pages developers previously visited with methods in their code by matching keyword terms [129]. Instead of recommending previous web pages, Prompter generates search queries that proactively recommends Stack Overflow posts related to the developer's code directly in the IDE [111]. Libra runs in the browser and ranks Google search results based on the quantity of information they provide, and their similarity to other results and the terms in developers' source code [112]. Switch! uses an ontology categorizing the different kinds of resources developers use to recommend resources when developers switch applications [82]. One challenge with these recommendation approaches is knowing when, where, and how to recommend the resources so that they integrate with developers' existing tools and workflows without distracting developers from their task.

2.3 Improving Developer Workspaces

Prior work has found that workspaces are not always well aligned with the way developers actually work in practice; notably, that developers spend a significant portion of their time outside of the IDE [8], search for information across tools [133], and use sets of resources for their tasks and that managing these sets can be difficult due to developers' frequent task switching [44]. However, these results are based on observations of a small number of developers [133], workers in professions other than software development [44], or large numbers of developers but at the granularity of developers' individual actions which can be difficult to interpret prescriptively [8, 9, 91]. Additional work that holistically observes larger numbers of developers working on their tasks would provide the background necessary for researchers to understand how developers go from a task description to the low-level actions they perform. This background would enable researchers to identify the diverse ways developers work and the ways tools (do not) support this work. At the same time, detailed observations about how developers overcome obstacles when using their tools may provide insights into how to address these obstacles.

One specific obstacle developers encounter is locating information. Prior work has identified specific kinds of information that existing tools are unable to provide to developers since different pieces of information are isolated

in different tools [39, 132, 144]. While approaches that organize (e.g., [116]) and recommend resources (e.g., [82]), and integrate information across disparate resources (e.g., [23]) can help developers locate information, each approach creates its own representation of the developer's task context by hooking into and extracting interaction data from the other tools developers use. Research into the ways this contextual information can be shared between tools could make these approaches more available to developers by allowing these approaches to be incorporated into the tools developers already use.

Activity	**Description**	**[133]**	**[74]**	**[91]**	**[8]**	**[13]**
Reading	Examining information from resources	19%	-	70%	-	5%
Editing	Making changing to project resources	8%	16%	5%	29%	26%
Navigation	Browsing, searching, and locating information within and between resources	9%	10%	4%	22%	25%
Execution	Managing and using tools to execute actions	38%	10%	14%	9%	34%
Project Management	Activities and actions to manage the project	8%	-	-	2%	-
Communication and Ideation	Thinking about and discussing coding tasks	9%	15%	-	-	-
Other	Other development-related activities	10%	13%	8%	38%	9%

Table 2.1: Development activities identified in prior work.

Chapter 3

Sources of Software Development Task Friction

In this chapter, we establish the fundamental concepts necessary to address our thesis statement, in particular, we formulate the concept of *friction* in developers' workflows. Developers improve their systems by making changes that fix defects or add new features requested in their tasks. However, developers must first decompose the task description into concrete objectives, or goals, which they complete by performing actions using different applications in their workspaces. These goals motivate developers' actions and represent their intentions as they seek out and respond to feedback from their tools combined with their experience and observations. Unfortunately, actions are often spread across multiple applications making it difficult for developers to obtain, integrate, and use their feedback [72, 81, 132]. While prior work has examined developers' information needs across tasks [72, 132], for specific activities such as code comprehension [120], debugging [76], and testing [18], and at specific tools, especially the IDE [8, 91], little is known about the overall process developers use to decompose their tasks into goals and actions, and in particular, the challenges they face in doing so.

To identify these challenges, we first need to understand what developers are trying to accomplish—their goals. Developers formulate their goals based on the description of the bug, maintenance, or feature task they are assigned. They operationalize their goals by performing the low-level actions provided across the applications in their workspace [33, 148]. In the context of coding, [135] describes this process as stepwise refinement where developers decompose their tasks into sterotypical solutions (goals) consisting of fixed sequences of actions. For example, a developer working on a bug fixing task may have the goal of testing their changes. To turn this goal into a concrete sequence of steps they can perform with the tools, that is to *operationalize* this goal, they may have to switch to their shell application and run commands to build and start their program, and then switch to the running program and observe changes in behaviour.

This indirect translation from task descriptions-to-goals and goals-to-

actions can be mentally demanding as developers need to ascertain goals and ideate how they can complete those goals with the available tool actions. However, these actions often require developers to locate, integrate, and use specific information across applications. This separation between information and actions requires developers to remember information as they switch between applications and keep track of the windows and resources containing relevant information. As developers complete goals, they must also organize or close resources that are no longer needed to maintain a usable workspace. According to Green's Cognitive Dimensions [46], such a system is considered viscous as it requires developers to perform extra actions across multiple resources. We refer to these extra actions as *friction* because they reduce the velocity at which developers can accomplish their goals. In this chapter, we investigate friction through the following research questions.

Q1 How do developers operationalize goals into low-level actions provided by the workspace?

Q2 What kinds of friction do developers encounter when performing their goals?

We analyzed the screen recordings of 27 developers as they completed their tasks using cross-application workflows in two studies. We first observed 10 developers working on their own tasks and then an additional 17 developers in controlled settings as they addressed two real issues on two open-source projects. By considering discrepancies between the goals developers wanted to accomplish and how they were able to operationalize those goals as sequences of tool actions we were able to investigate how developers decompose a diverse set of tasks into goals, and the frictions they encountered.

This chapter provides insight into how developers decompose their tasks into goals and how they complete those goals in their workspace. We identify three design aspects of the workspace which cause developers to experience seven types of friction in their cross-application workflows. This work motivates our two subsequent studies examining ways to address frictions commonly imposed on developers by their workspaces to enable them to more directly accomplish their goals.

3.1 Methodology

To learn how developers decompose and operationalize their tasks into goals and actions, we adopted a mixed-methods approach using data collected from both fixed and user-selected tasks. Our approach was inspired by the Cognitive Walkthrough method where an analyst selects a specific task, determines an ideal sequence of actions, and identifies and reasons about trouble spots by tracing the mental process of a user [77]. However, we apply these steps to actual users rather than working through the tasks ourselves. Specifically, we collected and examined live-streamed recordings of developers working on their own tasks in their own workspaces to identify the kinds of unnecessary work developers have to overcome. We then conducted a controlled user study to understand how developers approach a fixed task, and the impact the unnecessary work has on the workflows developers use to complete the task. A high-level overview of our mixed-methods approach is captured in Figure 3.1.

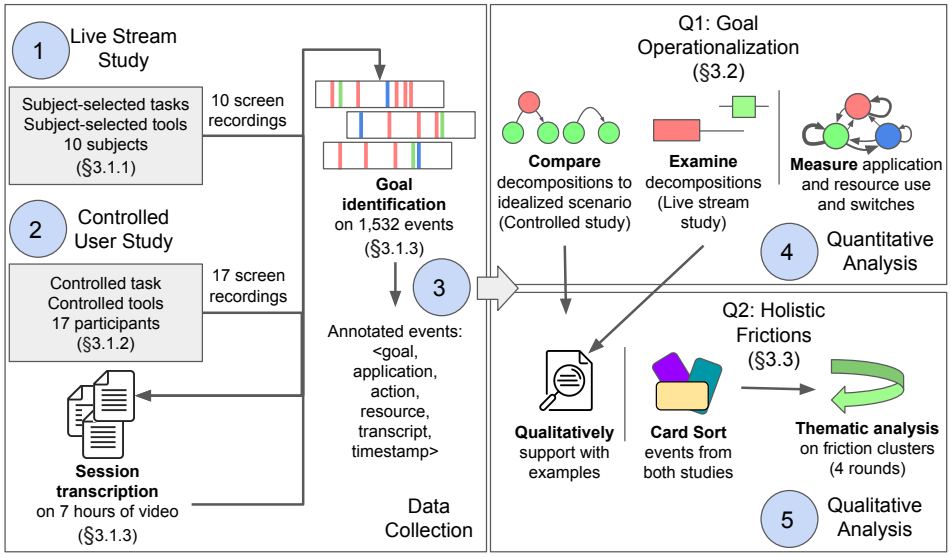


Figure 3.1: The mixed methods research approach used. The order in which the steps were performed are denoted ①–⑤.

3.1.1 Live-stream Study

Live streaming development sessions is a way for developers to share their expertise while building a sense of community. Prior studies have found these sessions are not rehearsed and illustrate developers' real contributions to software projects using their preferred tools [6]. Exchanges between the developers and their audience provide running commentary, similar to think aloud, describing what they are doing and why. We use live-streamed screen recordings of 10 developers (Figure 3.1, Step 1) to identify an initial set of challenges developers encounter when completing their tasks.

Subjects

The subjects[2] for this study were ten developers and a section of one of their recently streamed development sessions. Table 3.1 provides information about the presenters and their selected sections.

We identified developers and videos through an iterative search and review process. Videos had to be presented in English in a professional manner with a clear software development task which the presenter attempted to solve during the video. We excluded tutorials, Q&A and office-hour-type sessions, and videos designed specifically to demonstrate a feature or procedure. We started by examining the most recently published videos from members of the Live Coders team as they are required to present in a professional manner, write code regularly, and to have had at least five viewers in the 30 days prior to the review of their application.[3] Six of the developers were members of the Live Coders team. We identified the remaining four developers by searching for *live coding* videos on Twitch and YouTube. Our final set of videos capture a diverse set of tasks, environments, and program languages, sizes, and maturities.

Videos ranged from 1.5 to over 3 hours in length. During this time presenters worked on multiple tasks but always clearly described the task on which they were currently working, either orally or textually with issues or TODO items. We scanned through the videos identifying sections in which the presenter switched between multiple applications so that we could extract the maximum amount of information about the cross-application friction developers experience across a range of tools. We then isolated one section from each video that encompassed a full task.

[2] We label the people evaluated in the controlled study *participants* (P1..P17) and those from the live stream study *subjects* (S1..S10) throughout the thesis.

[3] `https://livecoders.dev`

Before finalizing our video selection we verified that the presenters had professional software development experience using the information provided on their LinkedIn profiles. The ten presenters (9 male, 1 female) each had at least one year of development experience and an average of 12 ± 7 years.

Table 3.1: Selected tasks from live-streamed sessions. Developer (Exp)erience listed in years.

Sub	Exp	Role	Time	Task Description
S1	7	Senior Developer	14m30s	Alter table sorting behaviour.
S2	10	Senior Developer	9m28s	Stop streams when window is closed.
S3	25	Technical Trainer	27m45s	Create a new Java Spring project.
S4	15	Cloud Advocate	15m10s	Understand Twitch service response.
S5	16	Senior Developer	7m55s	Release new version of Azure plugin.
S6	10	Developer Evangelist	26m49s	Create serverless blog search feature.
S7	15	Outreach Manager	17m00s	Migrate ASP.NET button to Blazor.
S8	15	Developer Advocate	6m32s	Refactor and deploy REST endpoint.
S9	6	Developer Evangelist	19m02s	Obfuscate tokens in log output.
S10	1	Developer Evangelist	16m26s	Create README and LICENSE.

Projects and Tasks

We observed subjects working on both personal and open source community projects. S1, S4, and S8 worked on bots that respond to commands issued by viewers through the Twitch API. S3 created a new Java project to compute stock options, S6 implemented a website search feature using Azure's Cognitive Search API, and S10 worked on a pull request tracker. S2 worked on the Mozilla Firefox browser project, S5 worked on C# Make integration for Azure DevOps, S7 worked on a project to recreate WebForms

components in Blazor, and S9 worked on Twilio's CLI for deploying serverless functions. The specific tasks each subject worked on during the selected section of video are shown in Table 3.1.

3.1.2 Controlled User Study

Our second study sought to gain insight into how developers decompose and operationalize known fixed tasks (Figure 3.1, Step 2). We used fixed tasks for all participants to remove task-induced confounds so we could confidently infer the goals and challenges of participants, while also allowing us to compare tool usage strategies between participants. Controlling the development task also allowed us to create a baseline workflow against which we could compare participants' approaches to gain insight into the the different ways developers breakdown their tasks, and the work involved in completing them.

Participants

We recruited a total of 17 software developers (3 female, 14 male) through personal contacts and recruiting emails. Ten of our participants were professional software developers while the remaining seven were computer science students (one undergraduate, six graduate) experienced in software development. On average, participants had 15.1 ± 10.0 years of programming experience and 8.2 ± 7.3 years of professional experience. We compensated each participant with a $20 gift card.

Projects and Tasks

We selected two bug fixing tasks from two real software systems that participants could complete within the 30-minute study. We identified the tasks by manually inspecting the recently opened issues of GitHub repositories labelled with the `beginner` topic. The first bug was reported on a 183 KLOC Java project called *TEAMMATES*.[4] The second bug was reported on a 230 KLOC PHP project called*Kanboard*.[5] We selected these bugs because the authors included suggested fixes that were simple to understand and test while still complex enough that participants had to use their tools. We piloted the tasks with an experienced developer and made minor adjustments to simplify the task descriptions.

[4] `https://github.com/TEAMMATES/teammates/issues/9545`
[5] `https://github.com/kanboard/kanboard/issues/4213`

Method

We provided participants with a laptop configured to track resources in popular development applications: IntelliJ, Visual Studio Code, Google Chrome, Firefox, and Terminal. The projects were pre-configured so participants would be able to work on the tasks immediately. We recorded the screen and took notes while participants were working.

We structured the study into three phases (OP1–3), with participants switching tasks partway through to simulate the interruptions which developers routinely encounter in their day-to-day work [44, 89, 106]. Before starting the study, we gave participants an overview of the two projects and tasks they would complete. We told them that they should try to have the two tasks done in 30 minutes and that a task was considered complete once they had opened a pull request with any changes they felt solved the bug. To allow us to compare workflows, we used the same phases in the following order for all participants.

OP1 Reproduce TEAMMATES bug. We instructed participants to begin by reproducing the TEAMMATES bug which was related to an improperly positioned button. After eight minutes, we interrupted participants and asked them to switch to the next phase.

OP2 Reproduce and fix Kanboard bug. We asked participants to work on the Kanboard bug which was related to the sizing of table columns, and to start by reproducing the bug. Based on our pilot, we provided steps to reproduce to help participants understand the bug more quickly. Participants were given sufficient time to open a pull request with their solution, before continuing to the next phase.

OP3 Fix TEAMMATES bug. Participants were asked to solve the TEAMMATES issue they started in OP1. After completing the task (2 participants), or a total of 30 minutes had elapsed, we asked participants to close their applications and stop working.

Idealized Workflow

The controlled nature of the tasks enabled us to create an idealized workflow consisting of the minimum number of actions developers would have to perform. Table 3.2 provides a high-level overview of the goals a developer could follow to fix the Kanboard bug (OP2). The table also includes the number of resources (i.e., files, shell commands, and web pages) they would use to perform each goal.

Table 3.2: An idealized workflow to complete `Kanboard` issue #4213. Numbers on the left of ▷ indicate the average number of resources used by participants in the user study. Numbers on the right indicate the ideal number of resources assuming no missteps, errors, or distractions.

		Application (Avg ▷ Ideal)		
Goal (Main Resources)		**Browser**	**Shell**	**IDE**
G1:	Learn what bug is about (Bug report)	4 ▷ 3	0 ▷ 0	0 ▷ 0
G2:	Start test environment (README, `docker`)	5 ▷ 0	5 ▷ 3	1 ▷ 0
G3:	Reproduce defect (Kanboard program)	16 ▷ 5	0 ▷ 0	0 ▷ 0
G4:	Find defect in source code (Program files)	4 ▷ 0	2 ▷ 0	6 ▷ 2
G5:	Review fix suggestion (Bug report)	1 ▷ 1	0 ▷ 0	0 ▷ 0
G6:	Fix bug (Program file)	1 ▷ 0	0 ▷ 0	2 ▷ 1
G7:	Check fix works as expected (`docker`, Program)	4 ▷ 1	3 ▷ 2	1 ▷ 0
G8:	Share fix as a pull request (`git`, PR form)	4 ▷ 1	10 ▷ 4	0 ▷ 0
	⟨participant average⟩▷⟨total ideal⟩	40 ▷ 11	20 ▷ 9	10 ▷ 3

During goals G1–G3, the developer learns about the defect and reproduces it. This allows them to confirm that the problem actually exists, they are able to observe the problem, and they have a procedure they can use to verify that their changes successfully fix the problem. Ideally, this would involve looking at three different web pages (the issue report, project README, and project documentation) in the browser and switching to the shell and executing three different commands (learned from the web pages) to build and start Kanboard using its containerized Docker-based test environment. Finally, they would launch Kanboard in the browser to manually verify that they could reproduce the defect.

Goals G4–G6 involve actually fixing the fault; it is here where the idealized workflow hides most of the task's complexity as the developer would need to find the affected file on their first try and fix it perfectly. The developer searches in their IDE and finds the file causing the defect. They revisit

the bug report to review the suggested fix and then switch back to the IDE to implement it.

During goal G7, the developer confirms that the fix worked. This involves rebuilding the `docker` image in the shell, relaunching the test environment, and switching back to the browser to verify that the fix was successful.

Goal G8 involves submitting the changes for the fix. First the developer performs four steps in the shell: checkout a branch, stage, commit, and push the changes to the remote server. Then they follow a link to GitHub's web interface to create a pull request for this change in the browser.

Fixing this simple fault needed changes to only two lines of code, but the developer had to decompose the task into eight goals which they would operationalize by switching between three applications nine times to access twenty unique resources a total of twenty-three times. The idealized workflow assumes the developer switched directly to the required resources and remembered all of the required information as they switched between applications and goals.

3.1.3 Data Preparation

Data from the two studies was processed through three high-level steps to prepare it for analysis.

Transcribing Sessions

In total, we collected more than 7 hours of screen recordings across the 27 subjects from both studies with an average duration of 16 ± 6 minutes. I transcribed events from the videos by recording the timestamp, resource identifier, application, and action performed each time a subject switched resources. In total, 1,532 events were transcribed from the screen recordings.

Separately from the transcripts, I also identified and recorded instances in which developers performed unexpected actions or encountered challenges. These included instances in which developers repeated actions or sequences of actions, switched quickly between windows, switched back and forth between resources, performed actions that failed (either explicitly with an error message or implicitly by not performing the desired operation), performed more actions than necessary to accomplish a goal, or when developers stated that they or their tools made a mistake or did not work as expected. It is from these observations that we derive developer friction (Q2).

Identifying Goals

After transcribing the screen recordings, our dataset consisted of lists of actions performed by each developer. To understand how developers decompose their tasks into goals and how directly they can complete these goals in their workspaces (Q1) we had to segment the transcripts into goals (Figure 3.1, Step 3).

When deciding where to segment goal boundaries in the transcripts, we considered each action in the context of its neighbours and how it was used by the developer. We examined the transcripts both forwards, following the actions the developers performed inferring what they were trying to achieve, and backwards, working from the results and inferring which steps were used to achieve them. We referred to the screen recordings throughout the process to ensure that we correctly interpreted the actions performed by the developers. We also looked for and used cues indicating developers' intentions when they were present. These included subject utterances and think aloud, e.g., "What we need to do now is implement actual searching." (S6), text they highlighted or copied, text they made visible by scrolling, keywords and search terms, and the resources they opened and closed.

Two researchers[6] (including myself) independently segmented three action transcripts from the live stream study establishing both goal boundaries and descriptions. In 16 out of 28 cases, the coders fully agreed and in 4 cases they were off by one action. In 8 cases one of the coders segmented the actions using an additional goal. Together with another researcher, we identified the aspects of the actions and cues that resulted in disagreements among the eight goals. Through this process of negotiated agreement [41], we were able to agree on how we should interpret actions in relation to the goals. However, identifying the precise location of goal boundaries was still a subjective process. To ensure consistency between the goals, I segmented the remaining transcripts following the agreed upon interpretation.

3.2 Making Tasks Actionable

Ultimately developers want to complete their tasks. But tasks can rarely be accomplished with one action. Instead, developers decompose their tasks into a series of manageable goals. Since each of these goals still cannot be directly executed by traditional workspaces, developers further operationalize each goal: they manually decompose their goal into a series of actions that

[6]The researchers are the authors of the first paper listed in the preface.

they can perform using their workspace. These two levels of decomposition (from task-to-goal and goal-to-actions) are not linear sequences: developers often encounter problems which can divert them as they solve new goals or devise new operationalizations. The decomposition process also requires explicit developer effort as they reason from their task through goals to the low-level actions they perform to complete their work. The workflows of the ten live stream study subjects are shown in Figure 3.2. We describe their task decompositions and goal operationalizations in the following sections.

3.2.1 Decomposing Tasks to Goals

Developers decompose their tasks into goals to provide intermediate objectives between the high-level task description and the low-level actions they need to perform to complete the task. This aligns with research on the cognitive process of problem solving in software development [148], and how people conceptualize their actions [142]. In this work we consider goals at the level of abstraction that developers would use to communicate with each other. We found that task decomposition happens on-demand and results in similar sequences of goals for developers completing the same task (Figure 3.1, Step 4).

Developers decompose the same task into similar sequences of goals. The controlled nature of the user study provided an opportunity to examine how participants decomposed their relatively straight-forward task into goals. We found that the sequence of these goals was generally similar among participants and the idealized workflow described in Section 3.1.2. Specifically, all but three of the participants ordered their goals in the same way: P8 interleaved goals while waiting for a command to finish, P10 choose to understand the proposed fix before reproducing the bug, and P17 choose to start the test environment before reading the issue.

However, not all participants used the same number of goals to complete the task. Nine of the participants decomposed the task into one *fewer* goals by using keywords from the suggested fix to locate the bug directly without a dedicated defect reproduction goal. Some participants also underwent *additional* goals when they sought a holistic understanding of the project before attempting to start the test environment (P1, P6, P7, P10, P11), and when they had to re-attempt failed goals, for instance after implementing the fix in the wrong code location (P13, P16) or attempting to reproduce the bug before starting the test environment (P12).

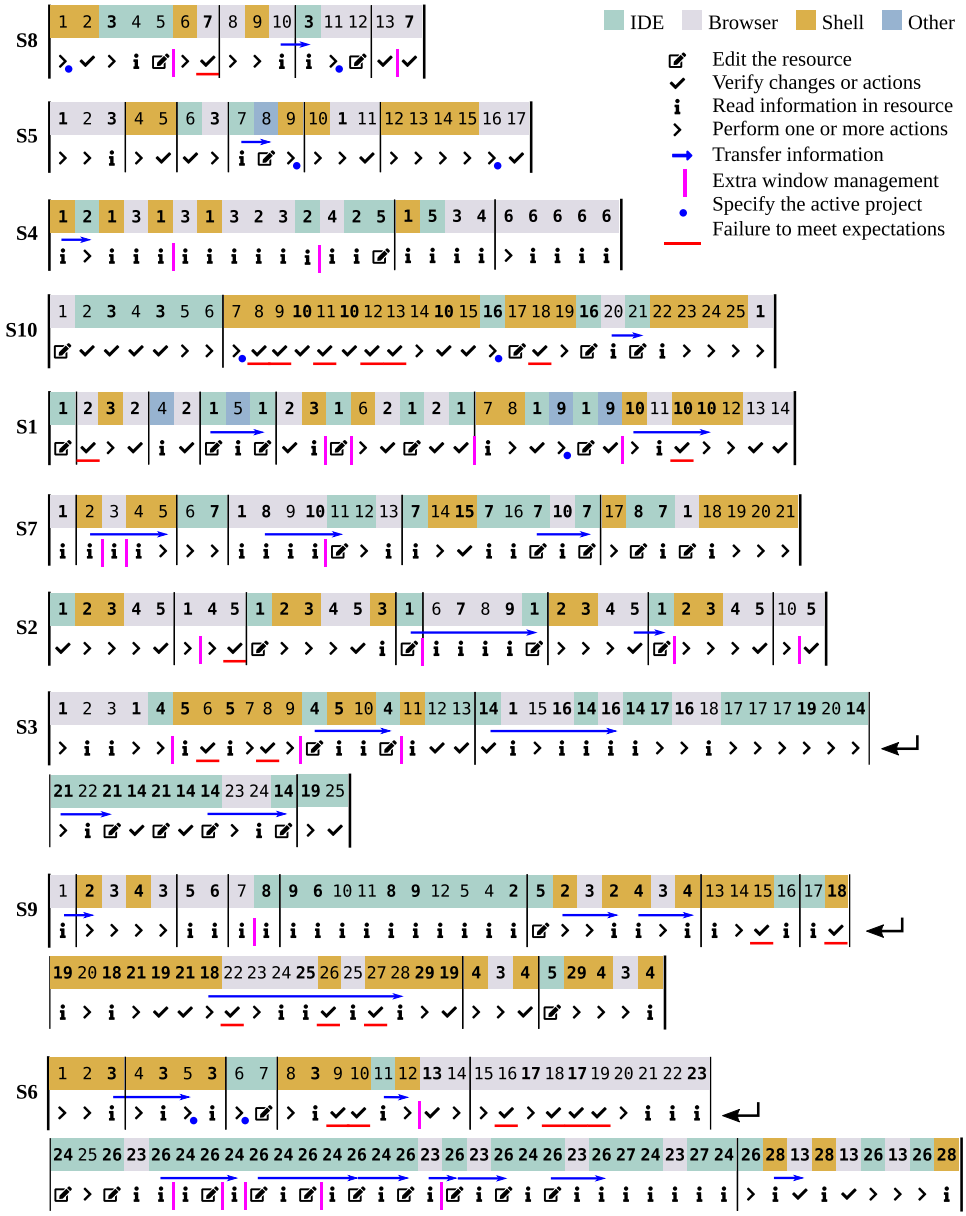


Figure 3.2: Workflows used by developers in the live stream study to complete their tasks. Vertical bars indicate goal boundaries. Numbers identify individual resources and the coloured background indicates the associated application. Revisited resources are shown in bold.

Developers formulate goals on-demand. Occasionally, developers encounter unexpected obstacles that they must overcome to continue making progress on their original goal. In these cases, they formulate new goals to overcome these obstacles. For example, when attempting to create a branch using `git`, P4 encountered an error stating she had provided incorrect flags. She responded that she knew "there was a way to do it" and proceeded to consult the help page, trying several variations of the command, before she was successful and could resume her original goal of sharing her changes.

There were also many instances of developers formulating on-demand goals in the live stream study. While debugging, S2 remarked "I didn't hit my breakpoint" and formulated a new goal to investigate whether the debugger "hits the unload event if you close a window." S3 formulated a new goal to investigate "why that [package version] is red? That shouldn't be red" when he noticed that the POM file in his Java project reported and error. He had to complete this emergent goal before he could finish creating his project and complete his original goal.

3.2.2 Operationalizing Goals to Actions

After decomposing tasks into goals, developers map these goals into actions that they use to interact with their resources. However, coming up with an effective mapping for a goal is not automatic: developers must consider what actions are available in each of their applications and how those actions can be combined across applications and resources to successfully operationalize their goal. Developers also have to handle discrepancies between their expectations and the actual result of their actions. We describe the factors affecting developers ability to map and operationalize their goals below (Figure 3.1, Step 4).

Workspaces do not allow developers to directly express their higher-level goals. Instead of directly communicating their goals to the workspace, developers have to mentally map their goals to the low-level actions offered by the workspace. This process is not trivial as the low-level nature of actions means developers have to consider and use many actions for each of their goals. On average, participants used 8.1 ± 6.5 actions per goal, using a maximum of 36 actions for a single goal. The need to consider so many actions imposes additional overhead as developers need to know how the actions can be effectively composed. For example, the excess switches in Figure 3.3 were the result of participants using extra actions to overcome incomplete mappings of their goals. This overhead is

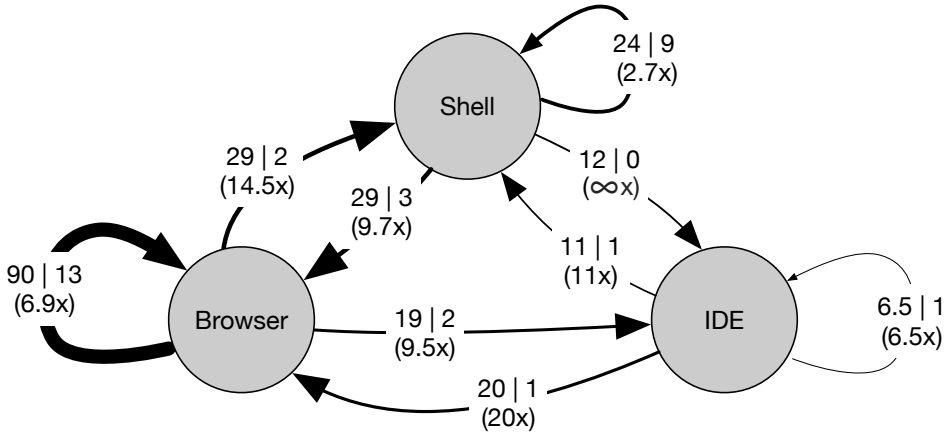


Figure 3.3: Resource switches. Numbers include the average number of times each kind of switch happened (left), the idealized number of times a switch would be needed (right), and the proportion of switches in excess of ideal (parentheses). Self-loops indicate that participants switched between resources within the application (e.g., from one web page to another), while arrows between nodes indicate that participants switched from a resource in one application to a resource in another application.

also evidenced by the actions that did not work as participants expected (shown using a _ in Figure 3.2).

Developers' knowledge and preferences of application differences affect their operationalizations. The applications developers use each have different user interfaces and set of actions. While some applications share similar interfaces at a high-level (e.g., tab-based web browsers), default key-bindings, parameters and syntax of text-based shells and even the interfaces of individual web pages vary. These interfaces each have individual strengths and weakness which developers must consider when operationalizing their goal. For example, P16 believed that it would be easier to run the project's docker commands from within his IDE rather than switching to the shell so he decided to spend extra time configuring his IDE's docker component.

Multiple applications often overlap in their functionality, requiring developers to choose among alternatives. For example, two of the user study participants (P8, P16) and 1 of 4 subjects (S10) in the live stream study that used version control during their task, used the IDE, while the remain-

ing participants and subjects chose to invoke git directly in the shell. Similarly, some subjects searched through their code using the GitHub repository search (S9), shell-based grep (P2), and using the IDE (all other subjects). These choices impact the number of actions developers have to consider when operationalizing their goal. For example, when committing changes, participants who chose to use the shell had to perform actions to see their outstanding changes and to verify they had been committed while this information was provided automatically in the IDE interface.

Operationalizing goals typically requires multiple applications and resources. Developers primarily use three kinds of applications: web browsers, IDEs, and shells. They used these applications to work with resources including code files, documentation, log files, and command output. Table 3.2 shows how participants used these applications to complete the goals of the task. Overall, participants in the user study used their browsers to view 677 (57%) resources, their shells to view 334 (28%) resources,[7] and their IDEs to view 174 (15%) resources, while subjects in the live stream study viewed their resources evenly across the three applications.

Beyond simple interactions with their workspace to switch between windows, developers had to perform actions in multiple applications across multiple resources to complete their goals. Figure 3.2 shows the applications, resources, and actions used by the subjects in the live stream study. The alternating colouring indicates that developers do not use applications sequentially but rather they need to switch between applications to interact with different resources while operationalizing their goals.

Developers must continuously locate and manage resources across applications. Developers need to locate, examine, modify, and manage many resources to accomplish their tasks. Managing resources is challenging because developers have to locate them before they can be opened, decide how they should be organized, and determine when they are no longer relevant and should be closed. This overhead increases with the number of resources used in a goal as the workspace becomes cluttered. Subjects in the user study used an average of 25.7 resources to complete the task and 3.4 resources to complete each goal. Subjects in the live stream study used an average of 18.3 resources to complete their tasks and 2.7 resources to complete each goal.

[7] Including the IDE-integrated shell.

To access their resources, developers have to frequently switch between different application windows and tabs (Figure 3.3). Over the course of completing their tasks, subjects switched between their resources an average 70 times (9 times per goal) in the user study and 34 times (6 times per goal) in the live stream study. Developers often switch to revisit resources, either to recall previously used information or to obtain new information from other areas of the resource. For example, subjects revisited the issue description to recall the suggested fix and to copy the exact keywords required to locate the source of the bug. Revisits are depicted in Figure 3.2 by boldface numbers. On average, subjects revisited resources 3 times per goal in the user study and 2 times per goal in the live stream study.

In some cases, only a small portion of a given resource is relevant to a goal. For example, subjects in the user study only used 2 of the 45 lines in the patched source code file, and only needed one sentence and 2 command descriptions from the whole project documentation. Developers have to spend time and mental effort getting to these pieces of information by first identifying the resource containing the information using a potentially unhelpful descriptor (e.g., a title or file name) and then reading, scrolling, or searching the resource to find the relevant content for their goal.

Information flows between applications and resources. The information developers need is spread across multiple resources and applications. This means that developers have to manually seek out and move the information between resources. They do so by using copy and paste or recalling the information from memory (sometimes with the help of the workspace; e.g., auto-completing directory paths).

Figure 3.2 shows instances where it was apparent that developers sought out and moved information between resources verbatim. Developers moved information under two scenarios. In the first scenario, developers used information they observed in a previously viewed resource in a later resource. This is the case when the arrow starts and ends on different resources. For example, S6 used information (directory names) from the output of `ls` to recall and change into his project's directory (depicted by the arrow from resource 3 to resource 5). In the second scenario, developers realized they needed some information in a particular resource and had to perform actions to obtain the information before moving it into the original resource. This is the case when the arrow starts and ends on the same resource. For example, when configuring his project to use the latest version of Java, S2 had to leave his IDE to find the installation path and replicate that path in

the configuration dialog.

Developers also move information implicitly across resources. For example, S4 used the property name of an object to inspect debug output, understand the structure of the object by searching for the name in online documentation, and to follow a hyperlink describing the property in detail. We do not show these implicit movements in Figure 3.2 since they require interpretation of the information and where it originated (i.e., the information may be abstract and not directly visible in the resource). It is not clear, for example, how S4 knew which property to investigate based on the resources he previously used.

Discrepancies between expected and actual results of an action cause additional actions and adjustments. The actions developers perform do not always work as intended. Figure 3.2 shows the actions that failed during the live stream study (using a _).

The most common failure encountered by developers was due to invalid shell commands (48%) (S6, S9, S10). Developers also encountered failures identifying the resources that contained the information they needed (26%) (S3, S6, S9) and when their actions behaved differently than expected (17%) (S1, S2, S3, S9). For example, S1 accidentally aborted a commit when he used the wrong key sequence to exit `vi` while S3 had to guess an alternative parameter value when a previous attempt failed. Finally, there were two instances where developers attempted to verify a code change by observing the output but failed to ensure the necessary preconditions were met (9%); e.g., that the development server was running (S1) and that the required environment variables were set (S8).

Sometimes it is not apparent when an action did not work as expected. In these cases, developers have to explicitly verify the effect of their actions, which are shown as a ✔ in Figure 3.2. Sometimes verification requires only a single action (e.g., running `git status` to ensure the commit action committed the correct files), but other times it can be a time consuming process (e.g., when manually testing a program change). Of particular interest are cases where developers have to re-verify their changes, repeatedly performing the same sequence of actions. For example, S2 modified his code five times during the development session and each time had to perform nine actions across both the shell and browser to verify the change.

Q1 Summary

Developers use similar sequences of goals when decomposing the same task, formulating goals on-demand to overcome unexpected obstacles. Operationalizing goals to actions is an indirect and personal process complicated by the need to work across applications.

3.3 Development Goal Friction

To complete a task, developers decompose the task into goals and then operationalize these goals by performing sequences of actions across applications in the workspace. Since developers' goals frequently cut across applications and resources, performing these actions can be difficult and induce friction that manifests itself as extraneous actions and mental effort. In this section, we describe the frictions developers experience when interacting with their complete workspaces, an area which has been largely neglected in favour of more targeted studies of individual applications and tools.

To examine the friction that occurs due to the mismatch between developers' goals and the actions available, we analyzed the video recordings and transcripts from both studies using thematic analysis [26]. After identifying the goals of each developer, I recorded descriptions of the actions developers performed that did not directly contribute to their goal and organized these descriptions into instances of friction. Three researchers (including myself) then conducted four rounds of categorizing the instances of friction into three higher-level themes that describe the ways developers interact with their workspaces: translating, integrating, and accessing resources (Figure 3.1, Step 5).

We observed a total of 386 instances of friction (231 instances in the user study and 155 instances in the live stream study). A summary of the frictions and their distribution are provide in Table 3.3.

Table 3.3: A summary of frictions in cross-application workflows. *Cause* summarizes how developers encounter these frictions and *Workaround* exemplifies how developers manually overcome the friction. *Burden* contains number of observed instances.

		Burden	Cause	Workaround
Low-Level Actions	**Translation Friction**	Plan Workflows 59/386; 15%	Crucial information for completing goals is often not completely represented in a fixed resource (e.g., who last changed a file). Developers must devise workflows to get this information considering the available tools.	Developers complete their goals opportunistically either using tools that are readily available or those they are familiar with. They seek out and use feedback about their actions to update their planned workflows.
		Learn and Adapt 55/386; 14%	Developers must learn how they can achieve a goal using the available actions. They form expectations about their tools which may lead to errors and confusion if behaviour is not consistent between tools.	Developers use trial-and-error to learn (and relearn) the steps necessary to complete their goals in different applications. When a tool does not meet their needs, they often install new tools or plug-ins, or perform extra work to adapt it.

Continued on next page

Table 3.3: A summary of frictions in cross-application workflows. *Cause* summarizes how developers encounter these frictions and *Workaround* exemplifies how developers manually overcome the friction. *Burden* contains number of observed instances. (Continued)

		Burden	Cause	Workaround
Dispersed Resources	Integration Friction	Switch 94/386; 24%	Developers often use multiple resources per goal which they must switch between. Standard navigation tools require developers to implicitly maintain relationships between resources, differentiate within- and across-application switches, and do not provide sufficient cues for developers to switch accurately.	Developers use CTRL-tab and ALT-tab to switch between their windows and tabs or visual search by manually clicking through their open resources. This process requires repeated switches which are distracting. Switching may fail, forcing the developer to re-open the resource.
		Organize 40/386; 10%	Operating system windows open at fixed sizes and positions, regardless of why the window was opened or how the content will be used relative to other resources.	Developers manually arrange tabs and windows to put related resources closer together so they can use information from multiple windows simultaneously.
		Transfer 61/386; 16%	The information developers need for their tasks is spread across different applications and tabs. This information must be manually integrated in order for developers to accomplish their goals.	Developers preemptively copy information they will subsequently need, or later navigate to a resource containing the information they require. This is usually accomplished with copy-and-paste.

Continued on next page

Table 3.3: A summary of frictions in cross-application workflows. *Cause* summarizes how developers encounter these frictions and *Workaround* exemplifies how developers manually overcome the friction. *Burden* contains number of observed instances. (Continued)

		Burden	Cause	Workaround
Independent Applications	**Access Friction**	Navigate 45/386; 12%	Goals require developers to manually access related resources across applications. Developers must traverse different organizational structures starting from application-dependent fixed locations (e.g., the shell's default directory or browser home page).	Developers manually navigate from the fixed default locations to their project location by either inputting the location directly or by using application-specific navigation steps (e.g., using `cd` in the shell or browser history).
		Configure 32/386; 8%	Applications need to be in a specific state for developers to perform actions and access information for their goal. This state is fragmented across applications, sometimes obscure, and changes as developers perform actions for subgoals.	Developers often emit details about their environment in logging statements to record and surface their values. They perform extra steps to set and verify their current project across all of the applications they use.

3.3.1 Translating Goals to Actions

The information and actions provided by the workspace do not always align with developers' goals, forcing them to *plan workflows* that translate their goals into low-level actions supported by the workspace. This mismatch between the workspace and developers' goals means developers often need to *learn* how the available actions can be made to satisfy their needs, or *adapt the workspace* to better conform to their desired workflows.

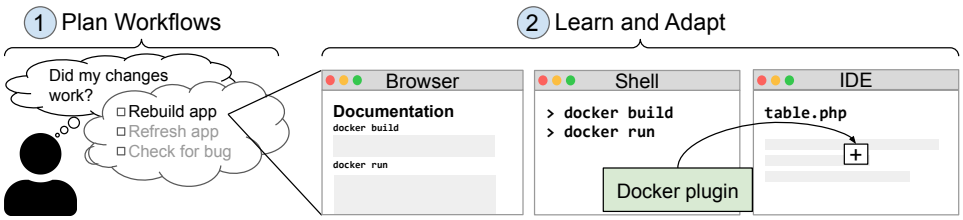


Figure 3.4: Developers must translate their goals into actions. They need to ① mentally decompose their goals into a plan the actions. This can require developers to ② learn the exact details of the commands or to adapt their tools to simplify the actions.

Planning Goal Workflows

The information developers need to accomplish their goals is often not represented explicitly in the workspace. Developers frequently need to devise workflows to answer high-level questions with the tools that are available to them. While some questions have answers that are available directly in the workspace, answering them still requires effort. For example, questions like "What branches do I have [locally]?" (S7) and "Where is [Java] installed?" (S3) can be answered by single commands, but identifying and executing the commands can interrupt developers' goals as they have to switch to separate applications and transfer the answers to where they are needed.

Unfortunately, many of these questions require more complex workflows for which developers need to seek out and synthesize multiple resources. One such case arose in the user study when subjects needed to know if their patch successfully fixed the issue. Since the workspace does not provide any way for developers to ascertain this information directly, subjects had to devise a workflow that included building their app in the shell, refreshing it in the browser, replicating the issue in the browser, and comparing the updated output with the screenshot in the issue tracker to actually answer their question (Figure 3.4, ①). When S3 got an error in his IDE he wondered "why is that [dependency] red?" He had to check online that he was using the correct version, and then had to refresh the dependencies, recompile the project, and invalidate the IDE caches. Ultimately, it was an error in his IDE but it took over seven minutes of attention to determine whether it was actually an error.

The information provided by workspaces can sometimes be ambiguous. When feedback is not as developers expect, they have to determine if they were expecting the wrong information or asking for the wrong informa-

tion. For example, subjects in the controlled study expected that their code changes would be automatically shown in the Kanboard web application based on their experience with other web frameworks. Instead, Kanboard required the subjects to relaunch the development instance after every change. The uncertainty of this expectation caused participants to wonder whether the page had actually changed and resulted in them spending extra time inspecting the page's rendered HTML code and comparing it with the screenshot provided in the issue tracker. S2, from the live stream study, was expecting that the breakpoint he set would pause execution allowing him to inspect the state of his application. When this did not happen, he had to determine what caused the breakpoint to not work as expected.

Learning and Adapting the Workspace

Developers form expectations about their workspaces that can lead to errors when they are incorrect. While errors help alert developers that their expectations are wrong, they do little to help developers effectively change their expectations. Instead, developers have to go through a tedious and time-consuming process of learning and correcting their expectations to conform to the workspace's conventions. For example, S10 expected the `touch` command would create an empty file but instead she received an error that the command is not available on Windows. She had to use six actions to learn an alternative method for creating a file. Similarly, P9 and P11 used eight and eleven actions, respectively, to learn how to move their commit to a new branch. Even experienced developers working in their own workspaces form incorrect expectations which they have to correct. For example, when attempting to install the latest Java version, S3 stated he "always forgets the commands" which caused him to run five unnecessary commands despite reading the help page three times. S9 expected the action `npm link` would create a binary but it took twelve commands, two Google searches, and reading a Stack Overflow and blog post to determine that the correct action was the very similar `twilio:link`.

Instead of adapting to the workspace's conventions, developers can change them through customization. This requires developers to be aware of their work habits and to seek out and incorporate the changes into their workspaces. For example, P16 digressed from his goal to spend more than 3 minutes configuring his IDE to run Docker directly so he would not have to work in the shell (Figure 3.4, ②). Similarly, P1 took time from his goal to alter the behaviour of his IDE by installing a VIM plugin while S8 configured his shell to use an advanced version of `cd`. Regardless of whether

developers choose to learn existing conventions or to customize them, they are both active processes that take developers' time and attention away from their goals.

3.3.2 Integrating Resources

Developers have to piece together information from multiple resources, such as source code files, issues, and Q&A websites, using various applications to accomplish their goals. Integrating information from multiple resources and applications can induce friction, especially when developers *switch*, *organize*, and *transfer content* between resources, requiring additional mental and manual effort. Figure 3.5 depicts these burdens for the reproduce and find defect goals of the user study task (Table 3.2, G3 and G4).

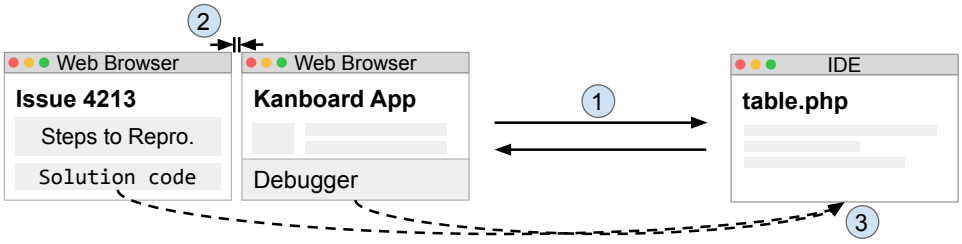


Figure 3.5: Resources are dispersed in windows across the workspace. To integrate these resources, developers manually ① switch between and ② organize windows to make information visible so they can ③ transfer it.

Switching Between Resources

The application and resource-centric nature of current workspaces commonly treats resources and applications as information silos offering only rudimentary support for the cross-cutting workflows used by developers. This *lack of cross-resource workflow support* often limits the simultaneous display of information and leads to developers frequently switching back and forth between resources to gather and integrate the relevant information located in various places. In both of our studies, developers frequently switched back and forth between resources. In the controlled user study for example, subjects switched an average of 13 times between the issue and the Kanboard program to apply the steps-to-reproduce, and S4 switched 10 times between an API response and its documentation to understand it.

This switching can incur a high cost, especially since it requires developers to get to the relevant resources again, often having to remember where they are and frequently resulting in "mis-switches"—switches to the wrong resource—that can further lead to distractions. Almost all developers in both studies had several mis-switch sequences (depicted in Figure 3.2 by |). Additionally, these 'switch cycles' often involve more than two resources, incurring an even higher switching and relocation cost. For example, S8 had to switch back and forth between six resources across three different applications to diagnose a bug. Another common switch cycle with multiple resources occurred when developers engaged in workflows to gather feedback for their changes. For example, after S2 made code changes in the IDE, he switched to the corresponding shell to build the project, then to the web browser to verify the changes. He repeated this loop five times to complete his goal.

Organizing Resources

Tasks and goals often require accessing resources in multiple applications and at the same time, resources often crosscut goals. This crosscutting nature of tasks, goals and resources can induce friction by requiring developers to manually organize their resources. In our studies, participants organized their resources by re-ordering application tabs (P4, S6), moving windows to virtual desktops (P1, P4, P5, P6, P17), positioning windows to make the contents visible simultaneously (P1, P6, P9, P11, P14, S4), or even using an additional browser application as a makeshift way to separate and organize the project's online documentation from the web app they were debugging (P13, P16). In some cases however, even the organizational mechanisms were insufficient, resulting in a high cost for relocating a resource. For instance, S6 had created a project but could not recall how it was organized within the filesystem: "I know I created a [project]...so where did I store it?" He had to navigate between five different directories to locate his project which took over a minute and caused him some confusion "I feel like I'm missing something."

Transferring Content Between Resources

In addition to switching between resources, developers frequently leverage and reuse information between resources (depicted in Figure 3.2 by →). This information transfer induces friction as developers need to either remember the information and apply it to the other resource, or relocate the infor-

mation manually using copy-and-paste, often also disrupting their flow of work. These situations occur frequently in any task, for example when developers copy and paste commands to run in the shell from some kind of documentation (e.g. S9), copy license information from files they worked on earlier (S10), or copy the branch name to use as a parameter in a shell git command (S7).

Transfer friction further increases when developers repeatedly transfer information and have to continuously relocate themselves or are transferring between resources that require complex switches. For example, S6 duplicated the structure of an existing source code file in a new code file by repeatedly switching back and forth between the two files, memorizing parts of the structure and recreating it in the new file. For each transfer the developer had to re-locate herself in the new file and recall what she was doing. Another common case across study participants was the transfer of the issue number into the commit message or a pull request description to link them. This transfer was performed by 14 subjects in the controlled and live stream study and required them to interrupt their current goal of committing code changes to switch to the issue tracker in the browser, find the issue, copy its number, and paste it in the shell. To work around the current limitations for transferring information and to avoid disruption to their workflows, we observed some developers preemptively copying information. For example, P7 preemptively copied credentials listed in the documentation although they were never needed and might have overwritten other relevant information in the clipboard.

3.3.3 Accessing Information

One of the common themes of developers' actions is to get to a resource to extract information or make changes. However, the workspace's siloed applications create friction making this information difficult to access. Even if developers know exactly which resource they need to accomplish their goals, getting to the resource in the workspace is often not straightforward requiring them to *navigate to the resource* across different organizational structures. Developers also have to perform actions to *identify and configure application state* as a prerequisite to performing other actions. Figure 3.6 depicts the burdens developers experience accessing the information needed to complete their goals across the siloed applications in the workspace.

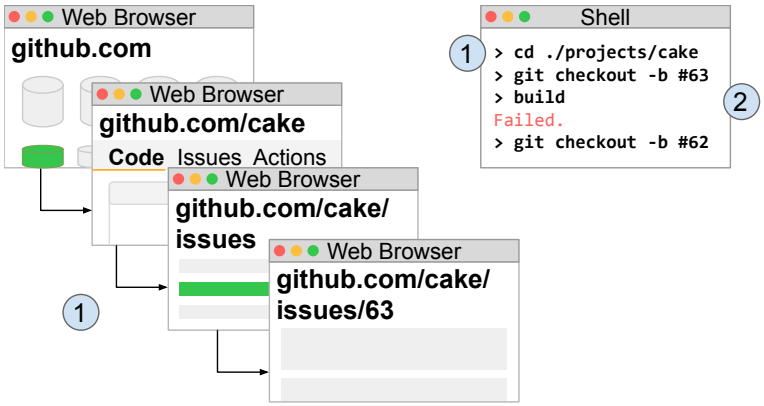


Figure 3.6: Accessing information requires developers navigate different organizational structures while maintaining appropriate state. Obtaining different kinds of information such as the issue number or build status requires developers to ① navigate different organizational structures. State, such as the active branch, can ② affect the behaviour of subsequent actions in unexpected ways.

Navigating to Resources

Resources are stored and accessed in independent structures that vary by resource type, application, and location (remote or local). For instance, to get the steps-to-reproduce contained in a task issue, the developer might open a new tab in a web browser window, go to her GitHub account, choose the 'Issues' tab and then the 'Assigned' tab within, scroll to the relevant issue, open it in the same tab and then browse the issue to find the reproduction steps, thereby navigating the structure of the tabs of a web browser and various (nested) structures within the GitHub service.

The *lack of uniform structures and mechanisms to access resources* requires developers to keep track of the different structures and mechanisms and correctly piece them together to navigate to the relevant resource. Developers frequently need to re-specify their project in different applications before they can access these resources. In our controlled user study, all 17 subjects navigated to the Kanboard project in both their IDE and on GitHub to access specific resources, and all but three also navigated to the Kanboard project in their shell, using an average of four `cd` and `ls` commands. Subjects in the live stream study had to navigate within even more applications, including Sublime Merge (S1), AppVeyor (S5), and AWS (S8) to access the relevant resources (depicted in Figure 3.2 by •).

Occasionally, developers go down the wrong path and have to go back or even switch applications. For example, after navigating to his project in his IDE, S9 starting looking for a necessary file by traversing the project's directories. He quickly gave up on this strategy and instead switched to GitHub where he navigated to the 'Code' tab. Using GitHub's code navigation feature, he traced through a series of method definitions to locate his desired file. He then had to manually navigate to the same file in his IDE so he could make changes. Just to locate a relevant file, S9 had to navigate different organizational structures in two separate applications using different navigational mechanisms.

Given the high number of resources and the high frequency with which developers access resources to complete their tasks, the induced friction can incur a high cost.

Identifying and Configuring Application State

Applications need to be in a specific state for developers to run actions successfully. For example, the shell's working directory determines where an action is run. However, the state of an application is not always apparent and remembering the state accurately can be difficult for developers. This is especially true when switching between applications and resources since they act as silos which fragment the state needed to complete a goal. As developers perform actions, they also inadvertently alter the workspace's state making it difficult to manage.

When the actual state of an application is different from what developers assume they encounter unexpected behaviour from their actions. This means developers either have to explicitly seek out the current state of their application or handle the behaviour of any actions they perform. For example, when the working directory of the shell was set outside of his project, P2 had to investigate the large number of irrelevant search results returned by `grep`. S8 had to trace through his program to identify the name of an environment variable he suspected to have caused an error. He then had to manually navigate through a web portal to locate and set the correct value for the variable. Had the environment variables been visible, this could have been resolved directly.

Q2 Summary

Workspaces often force developers to perform extra work to accomplish their goals. These extra steps introduce friction into developer workflows by requiring developers to adapt to their workspaces (translation friction), integrate information between resources (integration friction), and find and manage their resources (access friction).

3.4 Design Challenges

In this section, we examine the relationship between the workspaces' design aspects and the friction developers experience (see Figure 3.7 for an overview).

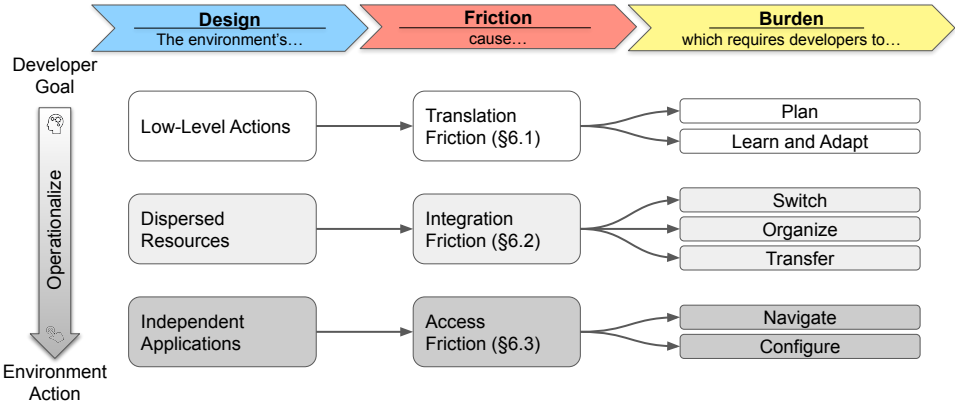


Figure 3.7: Relationships between the workspace's design and the burdens they impose on developers. Designs are ordered with burdens that are predominately cognitive at the top and predominately mechanical at the bottom to correspond with the process by which developers operationalize their goals into actions. For example, the workspace's *low-level actions* cause *translation friction* which requires developers to *plan* their workflows and *learn and adapt* the available actions—primarily cognitive burdens.

3.4.1 Low-Level Actions

To be able to complete their tasks, developers need to *plan* a workflow that consists of a series of low-level actions that the workspace supports. For example, installing a library requires developers to open a CLI, navigate to

their project directory, and run the installation commands provided on the library's web page. Checking out a code patch for review requires developers to commit their outstanding changes, pull the latest changes, and switch the the patch branch. Developers often resort to trial-and-error to construct these workflows, relying on feedback from their tools to *learn and adapt* their actions. Even once the developer has identified an appropriate workflow, they still need to recall and manually re-execute the actions for each of their tasks.

3.4.2 Dispersed Resources

Developers' cross-application workflows cause information to be shown in different places in the workspace. This dispersion of information can make it difficult for developers to integrate information related to their tasks. While developers are good at recognizing when and where they need to use this information, the process of obtaining the information is tedious and often disruptive as developers have to *switch* away from their current resource to manually obtain and *transfer* the information. One scenario developers frequently encounter is when they need to include a specific resource identifier (e.g., an issue number) when entering text (e.g., a commit message). For example, including the issue number in the commit message requires developers to locate and switch to the issue on GitHub, locate the issue number, and transfer the number into their message.

3.4.3 Independent Applications

Developers rely on a diverse set of isolated applications. Unfortunately, this isolation makes it difficult for developers to maintain a consistent state (e.g., working location, branch, environment variables) because they need to repeatedly *configure* this state in each application they use. Developers have to familiarize themselves with the different organizational structures and navigational mechanisms each application uses to represent and manage their state. Developers also need to manage state changes which occur inadvertently as they perform their actions. For example, when a developer switches projects, they have to manually set the project's location in each of their applications individually.

3.5 Discussion and Summary

An important finding of this chapter is that developers frequently encounter friction when trying to locate information across their applications. In the remainder of this chapter, we discuss this finding in relation to prior work, similarities and difference between the user and live-stream studies, and threats to validity.

3.5.1 Locating Information Across Applications

In this study, we oriented our investigation of developer's work around the friction arising from the actions developers take to complete their tasks, and, in particular, the friction associated with locating information. Information foraging theory (IFT) provides another perspective that attempts to formulate the decisions developers make when choosing where to navigate, using models original describing how predators choose their prey [110].

In IFT, developers forage for information in *patches* which, in this study, correspond to the resources displayed in application windows. Developers can either locate their information *within* a window, move *between* windows, or *enrich* the information shown within a window by performing actions. For example, a developer wishing to reproduce a bug would first read the bug report (within-patch) and then switch to the shell (between-patch) to build and run the program (enrich patch). If the program is a web application, the URL, shown in the run output, acts as a *cue* which prompts the developer to follow the *link* to the running application. The developer would then switch between the bug report and the application to follow the steps-to-reproduce (enrich patch) until they observe the bug in the window (within-patch).

Prior work examining IFT have limited the patches to individual applications (e.g., the IDE [107, 109]) or specific types of resources including source code files [75, 108], and version control logs [113]. Lawrance et al. found that the words used in bug reports can be used to predict which classes developers are likely to visit [75]. Within the IDE, Piorkowski found that misleading cues, information being scattered among patches, and the disjoint topologies used by different patches caused high navigation costs [108].

We extend this work by investing how these challenges manifest and impact developers in the novel context of their cross-application workspaces. Piorkowski et al. examined developers' foraging behaviour during a bug fixing task and identified six factors that made between-patch foraging difficult in the Eclipse IDE [108]. We observed three of these factors in cross-application workspaces: disjoint topologies, scattered information prey, and

misleading cues (false advertising). Consistent with this prior work, we found that getting to resources was costly due to the *disjoint topologies* used across applications forcing developers to navigate through different organizational structures (Section 3.3.3). However, we found this problem to be even more costly in cross-application workspaces because developers have to navigate from a base location (e.g., home directory) to their project in each application they use even before they can start looking for their desired information. As in the IDE, this information ends up *dispersed* across tabs and windows forcing developers to switch between them. However, these tabs can be open in one of many applications which can cause developers to mis-switch or walk through each tab and visually scan its contents (Section 3.3.2). This suggests that the cues provided in cross-application workspaces about the information contained in a window can also be insufficient or *misleading*.

In some cases, developers attempted to organize their windows anticipating revisits (Section 3.3.2). This is similar to *producer* effect proposed by [113] to describe the effort developers put into structuring their commits for future consumption by other developers working on the project. Surprisingly, we observed this even for resources managed entirely by the same developer and even for short-lived windows. Failing to anticipate these future needs led to expensive foraging to re-find resources.

Developers also adapted their workspace to obtain-goal relevant information (Section 3.3.1) which is known as *enrichment* in IFT. Unlike prior work which considers enrichment as an isolated activity (e.g., filtering a set of links), we found that it is often a complex process requiring developers to learn or customize actions (Section 3.3.1) and to manage (Section 3.3.3) and transfer (Section 3.3.2) the information they generate. In particular, we found that when developers were entering unstructured text (e.g., a commit message), they often had to interrupt themselves to seek out information from a structured information source (e.g., a list of issue numbers) which often required developers to perform multiple actions across tools, consistent with the observations made by [70].

3.5.2 Similarities and Differences Between Studies

We observed some notable similarities and differences in the types and frequency of frictions developers experienced in the user and live stream studies. In both studies, developers transferred content between resources a similar number of times. Both groups relied heavily on the shell to complete their tasks and experienced frictions associated with manually obtaining

goal-relevant information to overcome the lack of information provided by individual shell commands (e.g., explicitly listing the files in a directory or staged to be committed). Browsers were also used similarly between both groups to view documentation, manage their projects on GitHub, and test their web applications. However, subjects in the live stream study also used the browser to access other code-related services like build environments and cloud platforms, motivating further research into how these disparate services affect the development process.

Participants in the user study accumulated more windows over the course of the task as they followed links in the documentation and opened new shells. They also spent more time organizing these windows, likely the result of the smaller laptop screen and the number of windows they opened. Despite this organization, they still mis-switched frequently and resorted to either stepping through tabs individually or "thrashing" between windows seemingly randomly. We speculate the accumulation of windows was due to participants' unfamiliarity with the project and task: they may have been afraid to close resources that might be needed in the future, perceiving them as costly to re-open.

Subjects in the live stream study worked with smaller sets of windows, reusing the ones they had already opened (i.e., opening a link in the same tab or stopping a command in the shell to run another command) and were more willing to close resources. However, they spent more time obtaining goal-relevant information, for example, when trying to understand the behaviour of their programs or configuring their projects. They also spent more time managing their workspace (e.g., handling updates, installing project dependencies and tools, setting environment variables). These are indicative of the fundamental costs of working in a cross-application workspace and likely did not occur during the controlled user study due to the constrained nature of the task. In the live stream study, these frictions arose throughout the development process and subjects had to interrupt their tasks to handle them.

We believe that the observations from both studies are equally informative. The user study participants represent both novice and experienced developers who are working on new projects with unfamiliar tools and workspaces, requiring support to efficiently acquire and organize knowledge about their project. In contrast, the live stream study showed that developers familiar with their project and workspace still need support to obtain information and manage state across the different applications and services required during the development process. The fact that we observed the similar frictions in both studies, even when developers had optimized their

workspaces, suggests that the frictions may generalize across developer experience, tools, and workspaces indicating the need for better support.

3.5.3 Threats to Validity

The nature of the studies, observations, and analyses in this chapter gives rise to several threats to the validity of our findings.

Internal Validity. As with all qualitative studies, our findings are subject to the researchers' perceptions. To mitigate this bias, multiple coders were involved in segmenting the transcripts and identifying developers' goals using multiple cues from participants. Three researchers conducted multiple rounds of thematic analysis to refine the friction categories. However, it is possible that other researchers may have identified alternative frictions from the data.

In both studies, the developers knew they were being observed. This may have affected how they completed their tasks. In particular, participants in the user study may have felt they needed to complete the task quickly and rushed through the actions in an ad hoc manner. Subjects in the live stream study may have altered their actions since they knew their activities would be visible to an external audience. However, their experience presenting also helped to mitigate observation effects.

External Validity. It is possible that our results do not generalize to all developers working in all workspaces. In the user study we recruited developers through recruitment emails and personal contacts which may have introduced a selection bias. In the live stream study we collected a pool of candidate videos from developers listed on the Live Coders Team site and videos suggested by Twitch and YouTube. It is possible that by starting from a curated listing of streamers we selected videos that are not representative of streamers in general. To mitigate this, we included four streamers that were not part of the Live Coders Team. It is also possible that developers who choose to stream their development sessions do not represent developers generally.

In the user study, participants worked on tasks from two projects. While we selected tasks from projects with medium-sized code bases in a popular languages using conventional tools, it is possible that we may have identified different types of friction if we observed participants completing their own work tasks or additional tasks for other projects. While the live stream study mitigates this threat somewhat as those developers were working with

a diverse set of projects and languages, the tasks they selected to record may not be representative of all development tasks.

Ultimately, we tried to make our results as generalizable as possible by observing 27 developers, with diverse backgrounds and experience, working on real development tasks. They were using typical desktop workspaces and a broad set of tools and applications.

3.5.4 Summary

In this chapter, we have examined how developers complete their tasks, finding frequent misalignment between developers and their workspace. By watching 17 developers perform a controlled task and 10 industrial developers perform their own tasks, we observed developers frequently decomposing their tasks into high-level goals and then operationalizing those goals as sequences of low-level actions that they could manually perform. The misalignment between the high-level goals the developers want to perform and the low-level actions provided by their workspaces induces friction that impedes their progress. The most common forms of friction developers encountered were centered around locating information associated with accessing and integrating their resources across applications. These findings motivated our subsequent examination of approaches to help mitigate this friction.

RQ1 Summary

We identified three forms of friction developers encounter in their desktop workspaces that increase the effort required for them to complete their tasks. *Translation friction* arises from the unnecessary work required for developers to decompose their high-level goals into low-level actions. *Integration friction* arises from having goal-relevant information dispersed across the workspace in different windows/tabs. *Access friction* arises from independent applications that developers need to align with their task.

Chapter 4

Centralizing Resources to Support Revisits Across Applications

Developers repeatedly reference and navigate between a range of different resources such as source code files, bug reports, documentation, and online Q/A forums to complete their tasks [12, 106]. These resources are spread across different applications such as browsers, IDEs, and CLIs (Figure 3.3). This interleaving of resources, applications, and tasks can impede developers' access to the information needed for their tasks.

Re-finding resources is particularly challenging when developers resume an interrupted task since resources for multiple tasks can be open at the same time, even within a single application [92, 122]. Navigating to the wrong resource can cause developers to become disoriented or distracted from their task. With developers switching resources every 15s on average, these mis-navigations can occur frequently and have a meaningful impact on the developer [106].

To address these challenges, prior work has examined manual and automatic approaches for re-finding resources. Manual approaches support developers in explicitly linking or grouping resources (e.g., [23, 126]) yet require manual effort. Many automatic approaches use temporal and/or semantic features to recommend relevant resources (e.g., [82, 129]); however, they are often limited to web resources, fail to support developers' cross-application workflows and do not consider the current project the developer is working on for better tailoring of the recommendations. Finally, some approaches automatically group resources by task (e.g., [37, 66, 116]) yet either require developers to denote their tasks or require manual effort to adjust the groups due to inaccuracy. In addition, current resource recommendation approaches require the developer to open the resource in another application to interact with it, even if the developer only needs to look at it or copy and paste some code.

In this chapter, we introduce a prototype centralized tool, called Helm, which we designed to help developers more directly access resources required for their active project across applications. Specifically, Helm mitigates access friction by automatically suggesting project resources regardless of the application and whether the resources are open or closed. Our design goal was to help developers minimize window management overhead, avoid misnavigating between their windows and applications, and ease re-accessing previously opened resources.

Helm automatically associates resources with the developer's project, as they are working, and provides a single interface developers can use to revisit resources across common development applications including the shell. Helm's interface enables developers to preview and interact with resources to help them quickly look up information and perform simple actions in the context of their main task minimizing window management overhead; for example, in our prototype, developers can interact with a project's shell history to directly build and run scripts without having to manually open and configure new shell sessions.

In a controlled study with 17 developers, we found that Helm reduces the mental overhead associated with the access friction developers encounter when they have to navigate across applications in their workspaces. When resuming tasks, Helm supported developers re-establish the resources and application states required for the task by recommending 80% of the resources developers revisited.

With developers switching almost constantly between their tasks and resources, approaches to overcome the overhead associated with these actions could significantly improve developers' speed and accuracy when completing their tasks. Helm is a proof-of-concept prototype that demonstrates that approaches can be built to eliminate much of the redundant application- and window-based navigation necessary in current workspaces. By monitoring the resources developers access across applications and projects, developers can be assisted to quickly re-establish the context necessary to resume their tasks. While developers found that these resource-based recommendations helped them effectively locate information for their task, further work incorporating a resource-centric model into existing applications could further reduce developer's mental effort by improving the way developers navigate and interact with their resources.

4.1 Design: Interactive Resource Recommendations

Helm's primary goal is to reduce the manual effort developers need to expend to re-find and re-open resources. Further, Helm enables developers to forgo some switches by enabling developers to perform actions on the recommendations directly, without changing applications or resources.

When developers need to re-find resources, they activate Helm with a keyboard shortcut. Helm automatically infers the developer's active project (Figure 4.1D) and immediately recommends relevant project resources (Figure 4.1B), regardless of the application and whether the resource is open or not. Developers can refine the recommendations by providing search terms (Figure 4.1A), such as the resource's name. Helm's interactive preview (Figure 4.1C) facilitates quickly looking up information as well as performing simple actions without the developer having to switch and keep track of the resources.

4.1.1 Lightweight Resource Tracking

Helm tracks the resources developers use across various applications. Our prototype tool currently supports three primary applications: the shell, IDE, and browser. Each time a developer interacts with a resource in one of these applications, Helm records the *application identifier*, *timestamp*, *duration*, and application-specific data fields (Table 4.1). Helm can easily be extended for further applications through plug-ins.

4.1.2 Recommender Model

To provide relevant recommendations to developers, Helm (a) associates each visited resource with a project and (b) computes the relevance of each resource for the developer's active project using metrics based on prior work (e.g., [66, 82, 116]).

In order to integrate Helm into developers' existing diverse workflows, the model uses information that can be collected across common applications without making assumptions about the structure of developers' work.

Active project. To scope the recommendations for the project a developer is working on and avoid noise, Helm keeps track of the active project. Specifically, Helm monitors the *Project* attributes shown in Table 4.1. Whenever an attribute with a project name is identified, Helm auto-

Table 4.1: Resource attributes used in Helm's model. *Search* indicates that the attribute can and will be matched with the user-entered search terms. *Project* indicates that the project is extracted from the attribute. *UI* indicates that the attribute is used when displaying the resource in Helm's user interface.

Application	Attribute	Search	Project	UI
Browser	`url`	x	x	
	`title`	x		x
IDE	`file path`	x	x	x
Shell	`command`	x		x
	`exit code`			x
	`session`			x
	`working directory`		x	x

matically associates all resources that are visited in the following ten-minute time window with the identified project name unless a different project is identified. We selected a ten-minute window inline with prior approaches (e.g., [106]). Resources that are used outside a project time window are not associated with any project.

We focused on tracking projects since developers commonly work and switch between several projects. Also, projects are easier to detect and more clearly defined than more granular concepts like tasks, which could result in filtering too many relevant resources.

Resource relevance. For each resource a developer visits, Helm keeps track of the following metrics:

- Recency (rc): time since last resource visit;
- Frequency (fr): number of visits to a resource;
- Duration (du): total time that a resource was open.

Based on these metrics, Helm then computes the relevance of a resource as:

$$relevance = 0.8 \times rc + 0.15 \times fr + 0.05 \times du$$

The weights for the Helm prototype were chosen as an initial starting point to capture the structure of developers work: solving small localized

problems that require referencing the same subset of resources [33, 44]. For example, developers may refer to a library's documentation repeatedly when writing code that uses the library. Therefore, our model has a larger weight (0.8) for recency to ensure that resources related to the developer's current task are likely to be displayed even if the resources have only been viewed once or for a short time. We further discuss the choice of weights and its impact on the model's accuracy in Section 4.3.

Resource recommendation. When a developer opens the launch-bar, Helm computes the relevance of all previously visited resources, filters them to the currently active project, and returns the top ten resources based on relevance. Developers can refine Helm's ranking by supplying search terms and by changing the project. If a developer enters search terms, Helm ranks only those resources whose *Search* attributes (Table 4.1) match.

4.1.3 Interactive Recommendations

To both surface recommendations and enable them to be interactive, Helm employs a launch-bar user interface. This allows developers to open Helm with a keyboard shortcut to search, preview, and interact with resources. Figure 4.1 shows a Helm recommendation for a shell session where the developer is choosing to re-run their Docker-based test environment. Helm presents resources ranked by relevance and grouped by application, including files in the IDE, web pages, and shell sessions. The groups are ordered so that the most relevant resource overall is shown first.

When developers select a resource from the IDE or browser, Helm displays a preview of the resource, which the developer can use as a temporary window to recall information. These previews are interactive: the developer can interact with the page as if they had switched to the browser or editor and opened the corresponding resource. For example, developers can read or copy information from the resource without having to switch from their current resource to the recommended resource. For more in-depth actions, the developer can click on the resource and the resource will be opened in the corresponding application.

For shell resources, Helm groups the commands into sessions. In the preview pane, developers can select commands from the session, and Helm automatically generates a shell script with the commands set to execute in the correct working directory, which the developer can verify and execute. A concrete example is illustrated in Figure 4.1(c). The developer opened Helm and began to search for "docker". By the time they typed "doc", Helm

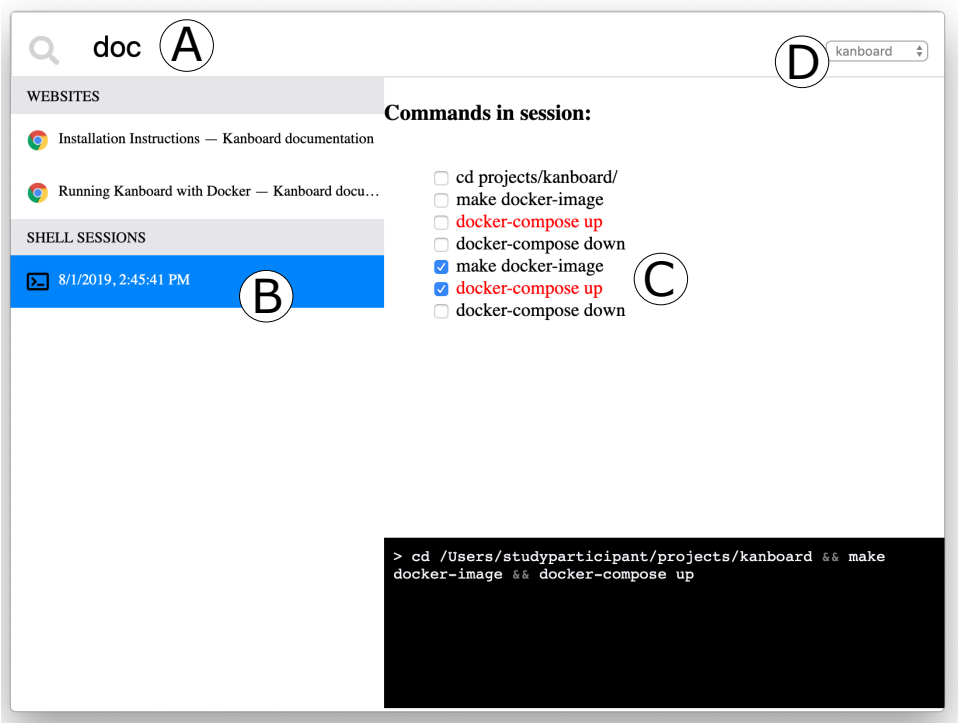


Figure 4.1: Helm's user interface. (A) Resource search bar (Table 4.1 lists the attributes that are searched). (B) List of search results, grouped by application (webpage, file (IDE), or shell session), showing the application icon and resource description. (C) Interactive preview of a resource. For webpages and files, it displays their content; for shell sessions, it displays the commands that were executed during the session. (D) A dropdown allows users to switch between projects. By default, the developer's active project is selected.

had already displayed their shell history, which contained the two relevant commands. The developer selected the commands from the history and Helm generated a shell script that the developer executed by pressing Enter (see bottom right). The test environment is now running: the developer did not have to open a shell, navigate to the correct directory, or remember the details of the commands.

4.2 Lab Evaluation

The focus of our evaluation was to explore whether Helm supports the various ways developers work, and to identify how Helm might be used in practice, including improvements needed for developers to adopt Helm in practice.

We conducted a lab evaluation of Helm with the 17 developers who participated in our controlled user study (Section 3.1.2). During the controlled user study, before using Helm, participants completed two bug fixing tasks; one each from the open source Teammates and Kanboard projects. Participants completed the two tasks in three phases: they started with the Teammates task for a short time (OP1), then switched to and completed the Kanboard task (OP2), and finally resumed and completed the Teammates task (OP3). We collected data about the resources participants accessed as they completed the two tasks. We used this data to evaluate the performance of Helm's recommendation model, and to seed Helm's recommendations individually for each participant during the lab evaluation.

For the lab evaluation, we provided participants with our Helm prototype. After demonstrating Helm's features, we asked participants to use Helm to complete a follow-up change to the Kanboard task they had completed during the controlled user study. The follow-up task was provided to participants as a comment on the pull request they had submitted. The focus of the follow-up task was to have participants use Helm to locate and update one of their previously edited code files, and re-run several shell commands to test and commit their changes. Once the participant completed the follow-up task, we had them report on the usability of the Helm prototype through a system usability survey, and elicited additional feedback about the approach through semi-structured interviews.[8] We allocated 30 minutes for the lab evaluation.

We successfully captured full interaction data from 14 of the 17 participants. Data for three of the participants was missing or corrupt. For one participant, the browser plug-in was not started correctly and the data was missing from their dataset; however, we included the data recorded for the other applications. We were unable to analyze the data of two other participants: in one case the data was not recorded, and in the other the database became corrupt before we could analyze it. However, all participants were able to try Helm and provide feedback. In the case of the missing browser

[8]The SUS survey and interview questions are available in the reproducibility package listed in Appendix A.

data, we described the web page view of the Helm interface to the participant, while in the case of the completely missing dataset, we ran Helm with anonymized data from a previous participant.

Overall, we captured 2,323 participant interactions with resources which we use to evaluate Helm's recommendation model. We also conducted a thematic analysis of the feedback from our interviews with participants to identify the benefits and challenges of using Helm in practice [26].

4.2.1 Helm's Recommendation Performance

To examine whether Helm would provide sufficiently relevant recommendations for developers to re-find the task resources during the lab evaluation, we first evaluated the recommendation model. We looked at each time one of the participants revisited a resource from the data we collected during the controlled user study and assessed whether the recommendations made by the model for that point in time contained the resource that was actually revisited by the developer. Of the total of 2,322 times a developer switched to another resource, 1,466 (63%) were switches to a previously visited resource. Participants primarily revisited resources in the web browser (69%), but also in the IDE (16%) and shell (15%).

For each of the 1,466 revisits by a developer, we simulated Helm's recommendation based on the prior switch history of the developer and the model described in Section 4.1.2. For each revisit, we then assessed whether the resource the developer switched to would have been displayed as one of the top ten results in Helm. Overall, Helm provided relevant recommendations for 80% (SD 7) of the 1,466 instances. In comparison, a random recommendation would only be relevant in approximately 11% of the instances, since the switch histories of developers contained an average of 93 (SD 62) unique resources for each switch. Broken down by application, Helm's model provides relevant recommendations for 83% of the resources revisited in the browser, 90% for the IDE, and 54% for the shell. Thus, for the majority of the 1,466 revisits Helm could enable developers to directly open the desired resource instead of having to locate and open them manually.

When developers switch from one application to another, the resource most recently used in the destination application might not be the desired one and the developer continues switching until they reach the desired resource. To account for these navigation sprees, we analyzed the switch histories and identified 29 cases in which participants switched between an average of 4 resources before reaching one they had previously visited and where they stayed for more than 2 seconds. For these 29 navigation sprees,

we again simulated Helm's recommendation at the instant of the first switch of each spree and found that Helm was able to correctly recommend the final resource 79% of the time. This indicates that Helm's recommendations could save developers multiple switches by enabling them to select their desired resources directly.

4.2.2 Developers' Use of Helm

To examine Helm's value in practice, we analyzed the quantitative data collected during the controlled user study and the qualitative interview data. In the following, we present the major findings and themes based on our analysis.

Helm supports re-establishing context more directly. In the lab evaluation, Helm supported participants in more directly reestablishing context relevant to their task at hand, either by revisiting resources or even by repeating commands in the shell. When starting the follow-up task during the lab evaluation, 13 (93%) of the 14 participants for which we collected the data, used Helm to open the browser to the correct GitHub project, and 10 (71%) even directly switched to the pull request web page using Helm. To complete the follow-up task, participants then had to re-open a file specific to the Kanboard issue. Despite all the files visited in between for the TEAMMATES and Kanboard issue, all but one participant was able to use Helm to re-open the correct file for the follow-up task on their first attempt (the other participant re-opened the file on their second attempt). In fact, Helm was able to recognize the switch to the Kanboard project based on participants opening the issue on the web in the beginning and, in turn, able to provide relevant recommendations for it.

In addition to supporting more direct re-visits of resources, Helm also enabled participants to re-execute multiple commands to re-establish project context. When verifying their changes to the code, 10 (67%) of the 15 participants for which we collected the shell commands, were able to successfully use Helm to compose and execute a script of multiple shell commands to re-build and start the project's docker container. When re-navigating to the Kanboard web app to verify the change, 6 (43%) of the participants directly launched the app's task view, enabling them to directly confirm their changes without any additional switches. An additional 3 (21%) participants launched the Kanboard webapp and then navigated to the task view.

Helm supports identifying project-relevant resources. Developers often work on multiple projects over the course of a day and have many windows and tabs open for each. Several participants mentioned that this makes it difficult to remember which resources belong to which project (P4, P8, P10, P11, P13). By keeping track of the project, Helm supported developers in identifying relevant resources when they switched between or within an application. Participants stated that this filtering of "*recently viewed stuff by project is pretty useful*" *(P5)* especially since "*it's really hard to keep track of all the tabs and all the things*" *(P12)* that are relevant for a project. Further, participants thought that "*it's helpful to have a list of commands that belong to a certain project*" *(P2)* and that "*it was nice to be able to go there and then just like pull up everything that was related to say the first project you were working on.*" *(P11)*

Helm reduces mental effort for switches within and between applications. In the interviews, the majority of participants stated that Helm reduced mental effort and helped in many situations in which developers have to otherwise remember and repeat many steps. Participants stated that Helm served "*as kind of an extended mental model*" *(P12)* and allowed them to "*backtrack and recreate situations*" *(P13)* which is particularly "*cool for those repeat[ed] tasks*" *(P5)*. While certain "*sequences of [steps]*" are "*almost in [developers'] muscle memory*" *(P2)*, both individual steps and sequences of steps can be "*forgotten*" *(P3)*, "*use a lot of mental energy*" *(P13)* and still take time and effort:

"*It's not like pain for me to write it in the terminal but still it takes time, much more time than just a couple clicks of the mouse [with Helm].*" *(P16)* In general, participants expressed that Helm eases the navigation within and between applications.

Helm helped developers *not* have to change to a resource. Participants frequently used or performed actions on resources without having to switch to them. Helm's ability to help users locate previously visited web pages was considered "*most helpful*" *(P12)* and "*more convenient actually than looking for [them] in the browser.*" *(P6)* Similar to what we found in the quantitative analysis of Helm's recommendation performance (Section 4.2.1), participants valued Helm's support in re-establishing context and repeating commands. For instance, participants stated for the shell, that "*Just knowing what directory you're in is actually half the battle. So having it take you to the right directory where the thing you're supposed to*

be doing something in is good. Cause you're usually like 'oh, that's 500 directories deep and somewhere'." *(P8)* and that "*[Helm] remembers where [the commands] were run so I don't have to 'cd' everywhere, that's good.*" *(P1)* Participants explicitly appreciated the fact that Helm was not only remembering resources but actually allowed them to execute past steps, such as re-running certain shell commands:

"*I liked that it was good to not only see what you did but that it was also runnable from the launcher.*" *(P5)*

Helm is useful and usable but could be more interactive. Overall, most participants were positive about their experience with the approach and expressed that it provided value, could enhance their productivity, thought that certain features were "*super useful*" *(P4)*, and were "*excited to see where [we] take it*" *(P3)*. The average score of 80 for the system usability scale indicates that developers also found Helm's usability to be good to great [10]. More specifically, participants (P7, P8, P13) liked how responsive Helm was noting that "*no loading time is very important when it comes to routine*" *(P13)*, and that having a separate UI is nice so they are not "*forced into a big, integrated thing*" *(P9)* but still have "*everything in the same place, which is really handy.*" *(P12)*

At the same time, participants had several suggestions for improvement in particular with respect to the information it provided, its transparency, and its interactiveness. Specifically, participants mentioned that Helm's UI could be improved by providing more information on resource labels (P5, P7, P8, P14, P17), showing better previews of the resources (P1, P3, P12, P14), also for instance, by including syntax highlighting of the last viewed location, and being more transparent of what is happening in the background (P13, P17). For instance, one participant stated:

"*I don't know what the tool was doing to associate all that activity together, and that's a problem because how do I leverage that tool. Like, I don't know how it really works yet but in terms of using the features of it: oh yeah, definitely. I would definitely use it.*" *(P13)*

For certain cases, Helm was also not the optimal solution. A few participants mentioned that Helm was slower and required more effort in some situations, especially when they already had many shell scripts defined for frequent workflows or because it was breaking some habits (P2, P13). This may have resulted from the lab condition where we asked participants to use Helm for all switches. While this condition allowed us to examine whether people could use the tool for several steps, it is not our expectation to have

users only use Helm to switch resources in practice.

4.2.3 Limitations

The data that we used to evaluate the performance of Helm's recommendation model was limited by the fixed tasks that developers completed during the controlled user study. While we used real tasks from open source projects, data from the field, with developers working on their own tasks over longer periods of time, may have yielded different recommendation performance results. However, the data from the controlled user study was sufficient to ensure Helm's recommendation model performed well enough for developers to assess Helm's overall approach while avoiding the privacy and support concerns associated with a field study.

The relatively short time that developers had to interact with Helm during the lab evaluation may have affected how they used Helm to complete the follow-up task, and feedback they provided about Helm. We provided a complete demonstration of Helm's features at the start of the follow-up task, and prompted developers to consider how Helm would fit (or not) into their typical workflows. A field deployment, where developers could actually use Helm in the context of their own workflows may yield additional insights into the benefits and challenges of the Helm approach.

4.3 Discussion and Summary

In the remainder of this chapter, we discuss how the independent design of applications in the workspace caused challenges when collecting information for Helm's recommendation model. We further describe how Helm contributes to prior efforts and discuss future improvements for Helm based on our observations and developer feedback during the study.

4.3.1 Information Sharing Between Applications

Helm shows that information about the resources developers access can reduce the friction of switching between different applications and re-navigating application-dependent organizational structures. However, one significant challenge when implementing Helm was simply collecting the necessary information about which resource the developer was accessing since the information provided by the workspace is limited. While the workspace provides programmatic access to the application names and window titles that are open, additional information, such as the location

of resources, requires plug-ins for every application the developer uses. Collecting and processing this information is also not straight forward; for example, we had to coordinate application focus events that were reported at different times depending on when the application executed the plugin, which made it difficult to accurately construct the state of the workspace.

As we discuss in Chapter 6, we believe this largely inaccessible information provides essential context which applications could use to help mitigate friction. In particular, resource locations contain easily extractable pieces of information about the developer's task. For example, Helm extracts the project name from resource paths and URLs, but other information developers commonly reference, e.g., ticket identifiers and branch names, are also embedded in the structure of resource locations.

4.3.2 Prior Efforts

Developers work with groups of resources associated with specific tasks [44], but re-finding these resources is often challenging because they are dispersed across different applications. Researches have examined approaches to help developers re-find resources both manually and through automatic recommendations.

Instead of recalling details about resources from memory to navigate among deeply nested window and file hierarchies, researchers have investigated different re-finding affordances including screen recordings [55], zoomable graphs of resource switches [146], and searchable key text snippets collected from developer's resources [47]. Researchers have also examined approaches to support developer's associative memory when re-finding by embedding links between resources e.g., code files and web pages [34, 43, 49, 126], and providing canvases where developers can link chucks of different resources together in a centralized view [3, 23, 36]. However, creating these associations is a manual process.

Automatic recommendations can help developers re-find resources more directly. These recommendations can be based on the resource's content (e.g., [129, 157]), or through pre-defined semantic relationships (e.g., [58, 82]). By using the task boundaries manually specified by developers, approaches can more accurately scope their recommendations [37, 66]. For example, Mylyn combines a degree-of-interest model with the developer's manually-specified active project to recommend files in the IDE [66]. CAAD automatically identifies the developer's task boundaries to group files across applications [116]. However, developers may need to manually adjust the groups as they do not always align with their actual tasks.

Similar to CAAD, Helm recommends resources based on developer interaction. However, rather than trying to show all resources, which is not feasible due to the large number of distinct resources developer rely on, Helm uses contextual information about the developer's project to focus the recommendations. Helm also enables developers to interact with their resources directly to look up information and perform small actions, without having to open and manage additional windows. At the same time, Helm enables developers to optionally refine the automatic recommendations to a specific task by specifying search terms, helping to keep them in-the-loop.

4.3.3 Future Work

This chapter introduced Helm, a prototype tool to examine whether a lightweight actionable resource recommendation approach could be useful for developers. From this initial investigation, several aspects of the approach require further investigation.

Model Performance and Weights. While Helm's 80% recommendation accuracy was effective for collecting representative feedback from developers about the overall approach during the exploratory study, further work is needed to determine whether these weights should be changed, or be adaptive to the developer as they work. For example, through a post hoc analysis, we found that recency dominated frequency and duration in our top-performing models, supporting our intuition from Section 4.1.2 that developers typically navigate between temporally-localized subsets of resources. Since the model is just one aspect of Helm, we based the metrics on prior work that also used recency, frequency, and time spent to determine relevance (e.g., [66, 82, 116]). However, we expect that additional information about the developer's task would improve the accuracy of the model, but accessing this information across applications represents significant work beyond the focus of our approach.

Recommendations vs. Search. Helm combines recommending resources with helping developers perform actions on these resources without switching to them. In the study, developers showed they were able to both locate relevant resources and interact with many of them directly within Helm. Several developers even thought that having Helm remain open to provide a list of resources would be a valuable reference while working. Surprisingly though, developers mainly relied on Helm's default recommendation list and did not search as much as expected. This may

be due to one of the primary threats to validity for our lab evaluation of Helm: The brevity of the study session (e.g., participants might not have had enough time to form a good mental model and search terms) represents an internal validity threat. Additionally, the current Helm approach does not support searching for content *within* resources, which may better align with how developers may expect search to work.

4.3.4 Summary

Developers frequently switch applications and resources as they work. Each of these switches is a manual process, and many of these switches are short due to either the small amount of information developers need from a resource or the small amount of work developers need to perform on the resource. Helm is a lightweight approach for recommending resources across applications, reducing effort required for re-finding and re-opening resources, which further allows developers to interact with these resources without opening them. By tracking the developer's active project, Helm supports re-establishing the resources and application state when resuming tasks more directly and with fewer switches. By considering why developers need to revisit a resource, Helm's interface helps developers avoid having to fully open the resource and incurring the extra friction associated with managing the resource. Through a controlled study we found that Helm could recommend the majority of previously-opened resources, and that developers were able to interact with many of these recommendations without having to switch away from their current application or resource, helping them to stay focused on their current task. These promising initial results indicate that recommending resources developers need across applications during a project is both feasible and can improve developer productivity. Further empirical examinations of the resources developers access could uncover deeper patterns beyond those explored and evaluated in our initial prototype.

RQ2 Summary

Helm's resource-centric approach helps developers to navigate and re-open the resources they needed for their tasks. Using contextual information about the resources developers access across applications, and by identifying the project the developer is actively on, Helm recommends relevant resources to the developer. In general, Helm helps to mitigate access friction by providing a centralized location where developers can directly access the resources required for their task, rather than having to navigate indirectly between applications, windows/tabs, and the filesystem.

Chapter 5

Locating API Signatures on the Web

In Chapter 3, we identified friction in developer workspaces, specifically access and integration friction, which makes it difficult for developers to locate information for their tasks. In Chapter 4, we examined Helm, a prototype approach that models the developer's active project and interactions with resources to recommend resources that developers have previously accessed. Helm addresses access friction by helping developers locate resources across applications from a centralized location instead of navigating through different organizational structures in the workspace. Helm use context about the developer's high-level interactions and resource metadata to recommend resources that developers have previously accessed. However, developers also need to integrate specific information across resources. Locating this information, specifically in new resources developers are not familiar with, induces friction in developer workflows since resources often contain extraneous information beyond what a developer needs for their current task [87].

In this chapter, we examine an approach to help developers locate specific information within their resources. Specifically, we focus on web search, which represents a concrete goal where developers explicitly seek out new information to solve their coding tasks [25, 40]. Locating information on the web introduces integration friction as developers have to frequently switch between their IDE and browser (Figure 3.3) [106], copy code terms into their searches [79, 150], and manage a large number of browser tabs as they assess different solutions [32, 92]. Even when a web page, such as a Stack Overflow (SO) post, contains the required information, developers have to read through the content to identify which solutions are present and how they fit into their code [56]. These extra steps affect developers' focus and increase their cognitive load [40].

One of the most common web searches developers make is to find information about application programming interfaces (APIs) [54, 149], which developers rely on to help them complete their development tasks [147]. We created a tool, called Scout, to help developers locate this API information

directly from their search results. Scout uses API signatures to succinctly represent search results making it easier for developers to locate information relevant to their task (Figure 5.1b). Specifically, Scout adapts the presentation of Google's SO search results so developers can more directly identify and compare alternative solutions by hiding non-essential and duplicate information.

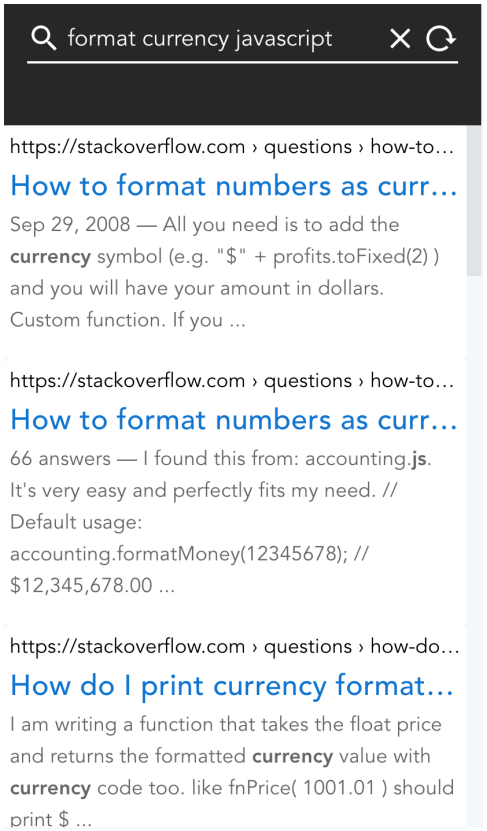
Scout extracts *API signatures*, consisting of a method name, class type, parameter type(s), and return type, from the SO pages in the developer's search results. These signatures are presented as focus points, giving developers an overview of potential solutions without having to examine each post individually. This overview encourages a top-down information gathering process where developers first identify the most appropriate signatures before deciding to examine detailed code examples and descriptions from the SO posts.

Scout is integrated into the IDE enabling developers to search directly from their code without losing focus on their current task. Using this integration, Scout automatically derives context terms, such as variable types and library names, from the code the developer is working on to improve the signature recommendations. When making a search, Scout suggests some of these terms to help scope the developer's query. Once the developer has made their search, Scout extracts the embedded API call signatures from each SO post in the search results. The top three signatures are presented under the title of each post, ranked by their occurrences in the results and overlap with the types in the developer's source code. The signatures are interactive, enabling developers to explore them in more detail by expanding minimal code examples so they can concretely assess the API's fit within their code (Figure 5.1c).

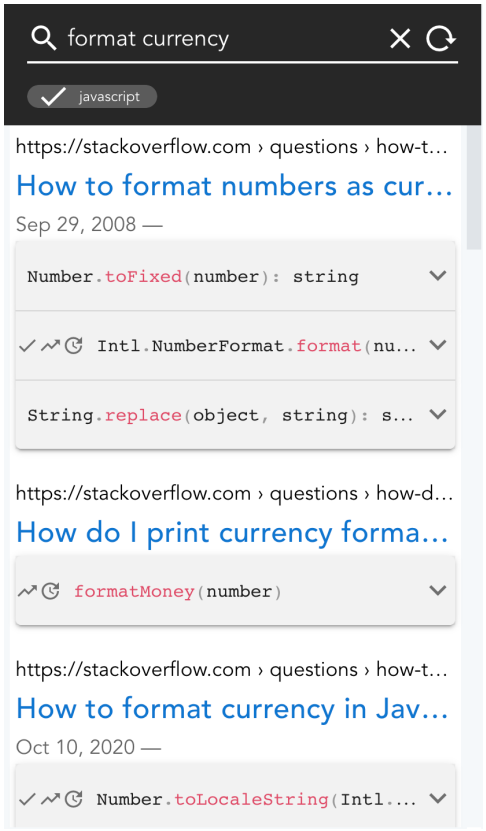
We conducted a controlled experiment with 40 developers following a paired design to evaluate Scout. In the experiment, we asked participants to complete six coding tasks, randomly assigning them to either our experimental treatment, which presented API call signatures (Figure 5.1b), or a baseline treatment which presented regular Google text snippets within the IDE (Figure 5.1a). For the study, we implemented Scout as a VSCode IDE extension that ran directly in participants browsers recording their interaction and feedback as they completed the study tasks.

Overall, Scout reduced integration friction by incorporating web search into the IDE using a tailored presentation of the search results. Scout provided developers with information for their task more directly, resulting in participants completing almost half of the study tasks directly using Scout's signature presentation, and significantly reduced the amount of posts par-

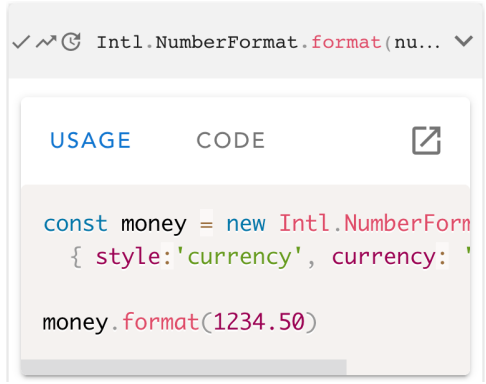
ticipants opened by 64%. Participants valued Scout's use of signatures to summarize their search results and found that the context terms Scout automatically inferred helped to significantly reduce the number of searches that they performed.



(a) Standard view with Google's text preview.



(b) Scout with API signatures.



(c) Expanded API call signature in Scout with class type (`Intl.NumberFormat`), function name (`format`), parameter types (`number`) and return type (`string`). Icons from left to right indicate whether the signature came from the *accepted*, *top-voted*, or *most recent* answer. The currently selected USAGE tab shows just the function call and the parent variable (`money`) while the CODE tab shows the full code example. Clicking the icon on the right opens the full answer inside Scout.

Figure 5.1: Search results interface.

5.1 Design

Up to 50% of developers' searches are to locate and obtain API information relevant to their current task [2, 52, 54, 79, 127, 149]. To perform these searches, developers usually switch from their IDE to a web browser, create a query, parse through the query results, go through the result pages to locate relevant information, and, once located, go back and forth between their code and the web browser to understand if and how the information fits [105]. These actions are extraneous to the task and detrimental to developers' focus and cognitive load [40].

Our objective is to direct developers to the API information most relevant to their tasks. Concretely, we aim to focus a developer's search and navigation towards task-relevant information, reduce the amount of content developers need to investigate, tailor the remaining information to the developer's working context, and present the results within the IDE to minimize cross-application switches.

We chose a progressive disclosure interface design which enables developers to first identify interesting APIs and then obtain additional contextual information to help the developer integrate the API into their code [31, 137]. We use API call signatures as focus points within the results of a search (Figure 5.1b) since call signatures are both compact and provide enough key information for developers to understand how they work [88]. Developers can expand the API signature to a minimal code example to obtain additional usage context (Figure 5.1c).

The progressive disclosure design resulted from an iterative design process informed by our prior observations of developers searching Google during live-streamed development sessions (Section 3.1.1). Our initial design used text snippets, extracted from SO answers referencing elements in the developer's source code, to explain the API. However, we found that prose made it difficult for developers to identify how to use the API (e.g., input and return types) and often required considerable space in the IDE due to the low information density of written text. In a subsequent design, we provided minimal code examples, displaying only the lines of code referencing the API related to the developer's source code, to make API usage more apparent. Unfortunately, even with this design, it was still difficult for developers to identify and compare APIs as the API calls were obfuscated by the other code in the example. Nonetheless, the design provided valuable context when developers wanted to use the API, so we included it in our prototype as the default view when the signature is expanded.

Our final Scout prototype is implemented as VSCode extension support-

ing the JavaScript programming language and focuses on retrieving API-related information from Stack Overflow as it is one of the most commonly used Q&A websites for developers [149]. Figure 5.2 provides an outline of the steps Scout performs when answering a developer's search query.

5.1.1 Augmenting Developer's Searches with Context

Scout uses context terms extracted from the code the developer is working on in the IDE's editor to (i) automatically augment the developer's search queries and (ii) rank the search results. Whenever a developer switches to Scout, the context terms are extracted from the Abstract Syntax Tree (AST) of the developers' active code file. Scout identifies the function the developer is working on (the *active function*) by using the cursor location in the code file. Scout retrieves the function's *class type*, *parameter types*, *return types*, and the types of any in-scope variables, along with the names of external libraries and any function calls made within the active function whose call chain originates in an external library. Scout retrieves the programming language from the file extension.

Scout automatically suggests the programming language, external library names, and library call context terms whenever a developer performs a query (Figure 5.2a). By default, these terms are added to the developer's query to tailor the search to the developer's code context. Adding these context terms to a developer's search query is motivated by prior observations that developers typically include terms such as the name of the programming language, frameworks, and libraries (e.g., HTML/JQuery) when searching the web [54]. The developer's search is ultimately performed using Google[9] with the `site:stackoverflow.com` domain filter (Figure 5.2b).

When recommending API signatures identified within the search results (Section 5.1.2), Scout uses the type context (class, parameter/variable, and return types) to rank the signatures based on how easily they could be implemented in the scope of the active function.

5.1.2 Identifying Relevant Signatures

To generate the API signatures from SO posts, Scout uses a three-step process (Figure 5.2c). First, the code blocks within each answer of a SO post are merged into a single virtual file which is then parsed into an AST using `ts-morph`,[10] a wrapper for the TypeScript compiler. When parsing,

[9] We use SerpAPI (`https://serpapi.com`) to avoid manually scraping search results.
[10] `https://ts-morph.com/`

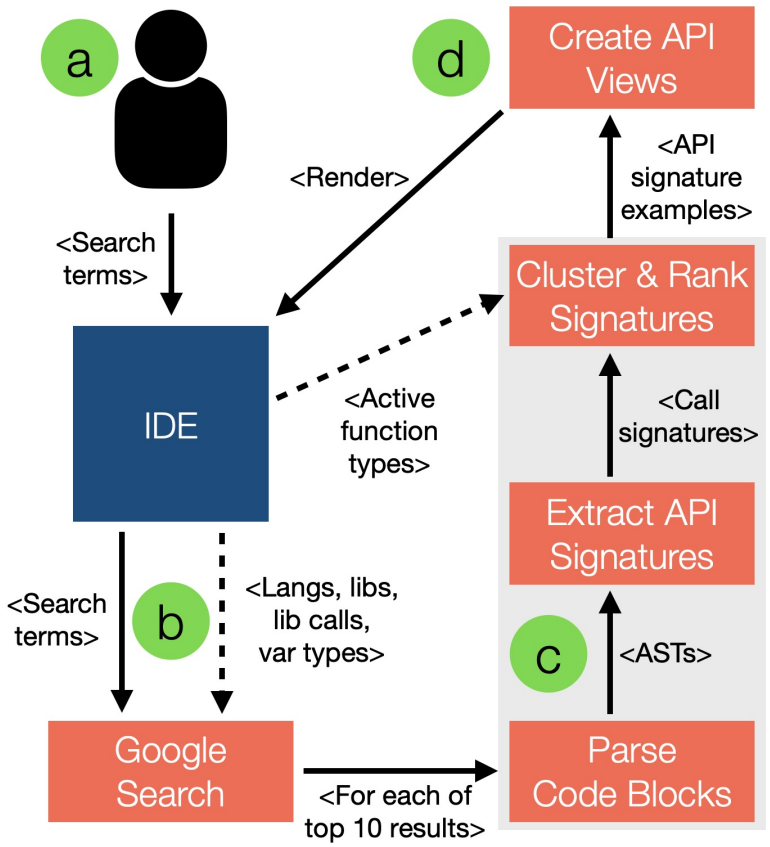


Figure 5.2: Scout's search process.

Scout assumes that the programming language in the code blocks is the same as that of the source code the developer is working on in the IDE; answers for which parsing fails are ignored. Second, Scout identifies the top-level call expressions within the parsed ASTs, excluding commonly used global calls, which, for JavaScript, include `console.*`, `alert`, and `require`. Finally, Scout uses a best-effort approach to resolve the class, parameter, and return types for the signatures based on the calls and surrounding example code in the SO answer. Scout infers the type from literal values used in the example. In cases where answers are incomplete (e.g., an undefined variable passed as a parameter), Scout extracts a partial signature omitting the types that could not be determined (for parameter types, we substitute `unknown` to maintain parameter positions). For calls that are part of the language, Scout further abstracts any literal types using the compiler's built-in type

definitions.

Scout includes the top three API call signatures generated for a SO post, based on the signature's frequency within the post and by the number of types that overlap with the function the developer is actively working on (see Section 5.1.1). Types overlap when the signature's class type matches an imported type or any types in the scope of the developer's active function (e.g., variable or parameter), the signature's parameter types match any types in the scope of the active function, or the signature's return type matches the return type of the active function.

5.1.3 Integrating Results Within the IDE

Scout renders a succinct summary of a developer's search results (Figure 5.2d), providing the title of the post (textual description) and the top three API call signatures and code examples essential for understanding and using the APIs [99, 151]. In contrast to Google where the information provided by the summary varies (Figure 5.1a), Scout's signature summary consistently provides the required information (Figure 5.1b). Additionally, Scout prefixes the signatures with up to three icons (check mark, arrow, and clock) indicating the quality of the SO answers (accepted, top-voted, and most-recent, respectively) the signature comes from. The compact nature of Scout's signature representation, combined with the textual description, make the representation ideal for summarizing search results within the limited space in the IDE.

To help developers better assess the suitability of an API call and integrate it into their code, Scout also provides a minimal usage example when the signature is expanded (Figure 5.1c). Scout extracts usage examples from the highest voted answers that contain the signature. For the minimal usage example, Scout includes only the function call, source object, parameter values, and the return value of the call. Scout also provides the corresponding full code example in the CODE tab (Figure 5.1c) from the answer of the post, in case developers need additional information to understand the API call and its context. Finally, developers can open the full answer on the SO post within Scout by clicking on the signature.

Our approach aims to reduce the effort required for locating and integrating relevant API information into the code. Specifically, to avoid the back-and-forth navigation between the IDE and pages in a web browser, Scout presents the search results next to the developer's active code editor and focuses on the most relevant information to accommodate the limited space within the IDE. The proximity of the search results to the active code

file, and the focus on the relevant parts of the search results within Scout, are meant to make it easier for developers to understand if an API call is suited for their coding task while allowing the API call to be used directly.

5.2 Experiment

Scout aims to direct developers to the API information most relevant to their tasks by reducing the number of posts developers need to investigate. To achieve this objective, Scout uses API call signatures to represent the information in the search results, and context terms from the developer's source code to augment the search terms and rank the signatures. We investigate these aspects in comparison to a traditional Google search approach, focusing on the following research questions:

Q1 How do API signatures and automatically identified context terms affect developers' search for API information?

Q2 How does the presentation of API search results affect the way developers locate relevant information?

Q3 How effectively do developers complete tasks with in-situ API representations?

Q4 What is the developers' experience using Scout?

We conducted a controlled experiment with 40 participants comparing our signature-based presentation with standard Google search results to evaluate whether Scout effectively distills the essential information developers require to identify API solutions. To collect and analyze the data we used a complementary, mixed-methods approach integrating participants' quantitative interaction data and qualitative feedback [16]. When assessing significance, we use Welch's t-test, after verifying normality using the Shapiro-Wilk test, and report normalized effect sizes with Cohen's d.

5.2.1 Method

The controlled experiment followed a within-participant design in which participants completed tasks with two treatments: a control treatment that presented results directly from Google search and our experimental interactive API treatment. Participants were randomly assigned to one of four counterbalanced experimental blocks based on a Latin Square design, as

shown in Figure 5.3. The experiment was conducted in three phases across the six independent tasks described in Table 5.1.

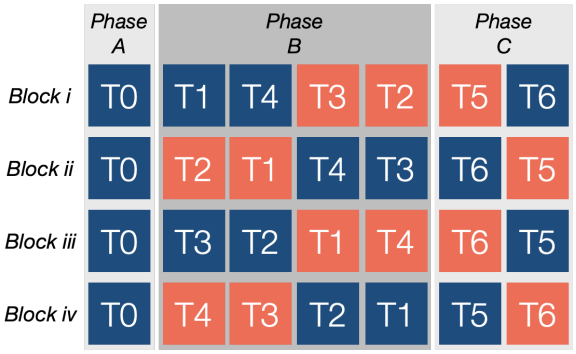


Figure 5.3: Randomized blocking for the tasks T0–T6. Dark tasks: Scout treatment; light tasks: control treatment.

Phase A: Training. At the outset of the experiment, we introduced participants to Scout through a guided walkthrough where they completed a mandatory tutorial task (T0).
Phase B: Controlled queries. In this phase, we asked participants to solve four independent tasks (T1–T4), two with the control (Google search) and two with the experimental treatment (Scout). While the task and treatment orders were randomized, treatment blocks were always consecutive to reduce disorientation caused by changing the interface too often. In this phase, we suggested an initial query for each task so that participants could focus on the presentation of the results without the difficulty of choosing effective query terms [67, 114, 115]. We constructed the initial queries based on the most common query format used by developers, consisting of the language, a verb, and compliment terms, and verified that the results contained a solution for the task [54]. The initial query also served as a control ensuring that participants were given the same initial set of search results under both treatments. However, participants were free to make additional searches if they desired.
Phase C: User queries. The final two tasks (T5–T6) simulated scenarios where participants needed to formulate and refine their search queries, as they would when completing their own development tasks. As with Phase B, participants were able to revise and search as many times as they thought necessary to find information that could help them complete their task. The initial treatment in Phase C was aligned with Phase B to reduce treatment disorientation.

After each task in Phase B and C, participants were given a short two-question survey to reflect on the task and to take a break before continuing. While each task could be solved in multiple ways, the simplest solution for each involved API calls. Participants completed the tasks by implementing the task requirements in individual methods following a test-driven approach using the provided unit tests to check their solutions. Participants had to pass all unit tests in the time allotted for each task (Table 5.1) before they could progress. Participants were notified when one minute remained; if one or more tests were still failing once the time had elapsed, participants were prompted to immediately continue to the next task.[11] The one exception was the tutorial task T0, for which there was no time limit and participants had to pass all tests.

Table 5.1: Task descriptions. Suggested keywords include both context terms (in bold) and search terms (T0–T4).

Task	**Time**	**Description/*Suggested Keywords***
T0		Sum numeric property in object array. ***javascript*** *sum object property array*
T1	6 min	Clone array of objects. ***javascript*** *clone array*
T2	6 min	Sort array of objects by property. ***javascript*** *sort array of objects descending*
T3	6 min	Find object in array with a specific value. ***javascript*** *find item in array by value*
T4	6 min	Localize currency display format. ***javascript*** *format currency*
T5	7 min	Compute a start date given end date + interval. ***javascript moment***
T6	7 min	Create endpoint to serve a PDF report. ***javascript express path***

[11] However, participants could dismiss the prompt and continue working.

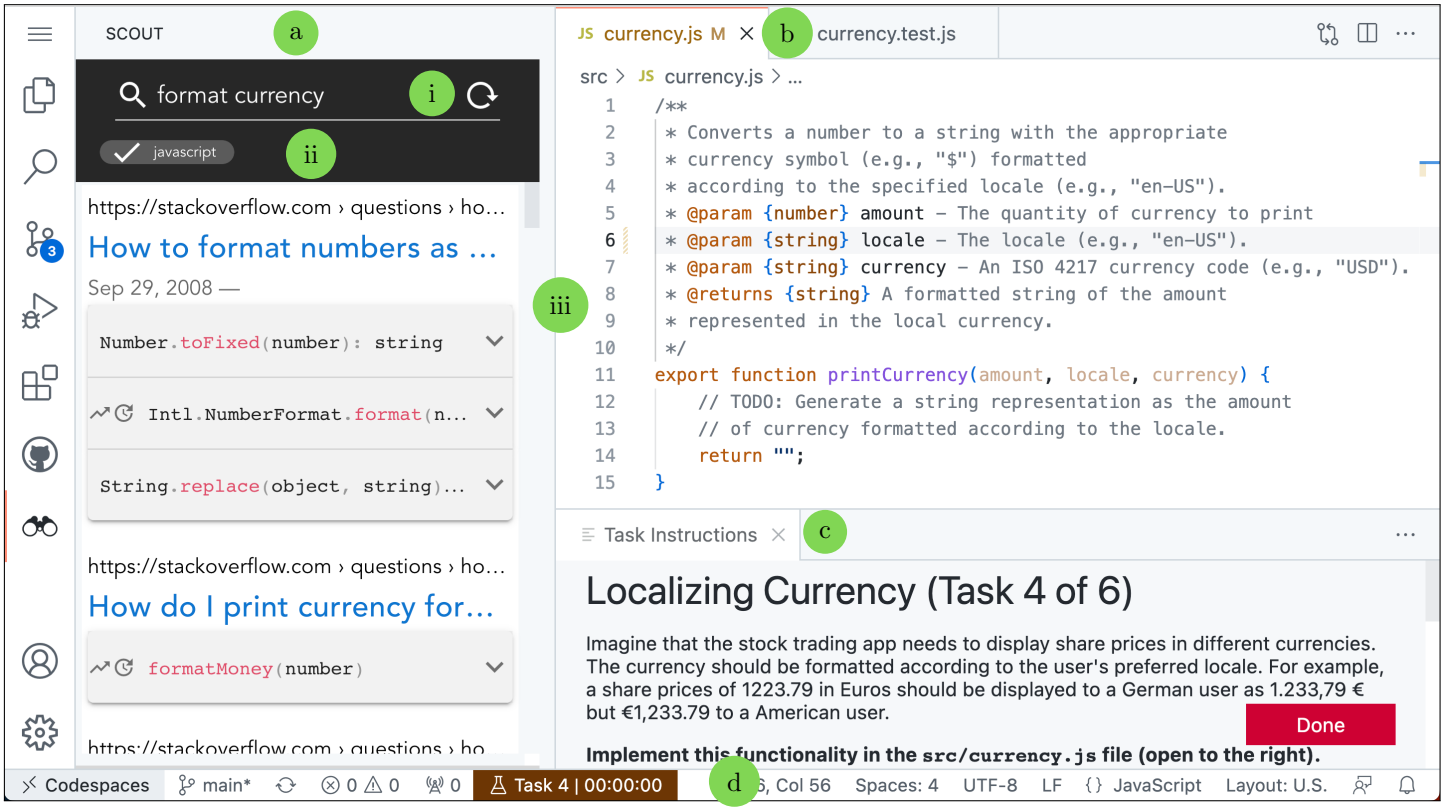


Figure 5.4: Online study environment. Scout is open in the left panel (a) where a participant has entered a search (i) which includes the selected context terms (ii). Underneath, the search results are summarized as call signatures (iii). The source code and instructions are shown on the right (b)–(c) with the time remaining in the status bar (d).

The study was designed to take between 60 and 90 minutes to complete, and to be executed entirely within the participant's browser. Participants were able to begin the study at any time but had to complete it within 180 minutes. To begin the study, participants followed an anonymous link to our online survey which provided details about the study including the number of tasks, and the overall time and JavaScript experience required. If the participant consented, they were randomly assigned to one of the four treatment blocks for the remainder of the study. We automatically provisioned a GitHub repository, started an online VSCode IDE (Codespace) instance, and provided participants with instructions to access this IDE in their browser.

The web-based IDE for the study is shown in Figure 5.4. The main Scout view is shown in Figure 5.4a; participants could enter query terms (Figure 5.4i), view or disable automatically-provided context terms (Figure 5.4ii), and view the API call signatures (Figure 5.4iii) adjacent to their code (Figure 5.4b). Instructions for each task were shown in a dedicated pane adjacent to the source code (Figure 5.4c). Participants could click the `Done` button once the tests passed (or the timer expired). Clicking `Done` replaced the instructions with the task feedback survey and a link to the next task. The time remaining was shown in the status bar at the bottom of the window (Figure 5.4d). Once participants completed the final task they were directed to an exit survey about their overall experience and demographics.

5.2.2 Participants

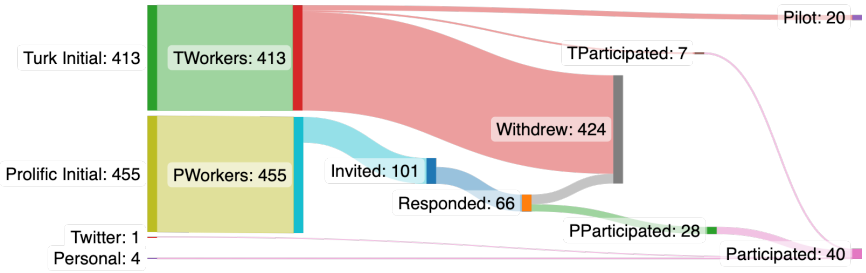


Figure 5.5: Participant recruitment process.

We advertised the study on a variety of online platforms including Twitter, Mechanical Turk,[12] and Prolific,[13] in addition to directly contacting

[12] `https://www.mturk.com`
[13] `https://www.prolific.co`

potential participants through personal contacts. Workers had to demonstrate their development knowledge by successfully completing a screening questionnaire to be eligible to participate. Figure 5.5 outlines our recruitment process for the study.

On Prolific, 455 workers had profiles listing appropriate development experience [125], and passed an established pre-screener which involved answering five basic development questions in a two-minute survey [35]. As we were not sure how many workers would actually complete the study, we decided to invite the top 100 qualified workers based on the number of studies they previously completed on the platform. We also invited one additional worker who requested to take part out of interest. Ultimately, we received responses to 66 of the 101 invitations sent. On Mechanical Turk, 413 workers opened our survey. Since Turk does not have a separate pre-screening and invitation process, workers had to complete the screening questions at the start of the survey before they could attempt the study. On both platforms, participants who did not complete the six study tasks (either successfully or by timing out) were considered to have withdrawn from the study.

Before conducting the full experiment, we took steps to ensure the quality of our experimental procedure. First, we piloted the study with 20 Mechanical Turk workers. We simplified the tasks so they could be completed in the allotted time, modified two survey questions to elicit more direct responses, and added checks to ensure that participants were actual developers and that they attempted all of the tasks. Second, we estimated the number of participants required to observe differences between the two treatments. We calculated that we needed a minimum of 22 participants, contributing 66 data points over the three tasks in each condition, using a t-test power analysis with a power level of 0.8, significance level of .05, and recommended Cohen's d effect size of 0.5 [38].

In total, we received 40 complete responses from 21 professional developers, 10 students, and 9 individuals who work with code in other capacities. Participants (31/40 male, 7/40 female, 2/40 transgender and nonbinary) reported an average of 5.8 ± 5.9 years of professional experience and 3.4 ± 4.3 years of JavaScript experience. The majority of participants came from Prolific (28/40) with the remaining from Mechanical Turk (7/40), Twitter (1/40), and through direct contact (4/40). Participants were paid $25.

5.2.3 Data Collection

The 40 participants completed 240 tasks in total. Participants were equally assigned to each of the four trial groups, resulting in 20 observations per

task (T1–T6) and treatment (control and Scout). We consider all of these observations in the following analysis, including 67 cases (34 control + 33 Scout) where participants exceeded the time allotted for the task. However, we exclude 11 cases (6 control + 5 Scout) where participants did not open the provided search interface and focus on the remaining 229 observations.

We recorded participants' interaction with Scout while completing the task including time spent on the task, searches made (including context terms), search results (including rank, processing time, and signatures), and navigation of the search results (including signatures expanded, results opened, and page scrolls). The data was stored in log files committed to participants' provisioned repositories and merged into a single database for analysis along with participants' responses to the post-task questions and the final survey. The study instruments and data are provided in Appendix A.

5.2.4 User Performance

In this section, we compare participants' searches, the SO posts and answers they viewed, and the success and time for completing the tasks across the 229 cases in the two experimental conditions.

Searching

Overall, participants made significantly fewer searches when using Scout, going down 19% from 1.6 to 1.3 searches on average ($t = -1.98, p = .025; d = 0.26$). In Phase B, searches decreased from 1.3 to 1.1 (15%) and from 2.1 to 1.7 (19%) in Phase C. One explanation for this decrease could be that the API signature summaries allowed participants to recognize relevant solutions faster so they avoided immediately making a new search. Another explanation could be that Scout's automatic addition of context terms to searches led to more suitable initial results requiring participants to make fewer refinements to their search queries.

To investigate the effect of Scouts' context terms, we examined the searches participants made during tasks T5 and T6 (Phase C, Figure 5.3). We consider the first search participants made during a task as their initial search and any subsequent searches as refinements to improve their results. For each task, we ordered participants' searches by time and manually marked keywords that overlapped with Scout's context terms to account for misspellings and abbreviations (e.g., js for JavaScript). Of the 152 searches participants made in total, 83 were under the control condition (with 39 initial searches, as one participant did not search) and 69 were under Scout

(40 initial searches).

Overall, we found that participants refined a smaller proportion of their initial searches under Scout when context terms were provided (14/40 vs 26/39). In the control treatment where no context terms were included, we found that 32/83 searches included a context term that would have been added by Scout automatically. Of these 32 searches, participants manually included a context term in 21 of their 39 initial searches and 11 of the 26 searches they refined. These manually entered terms were distributed across the names of libraries (60%), programming languages (41%), and API calls (2%) aligning with the terms Scout suggests. However, some of the terms Scout suggested did not always align perfectly, with participants removing a library term in 3 of the 69 searches.

When asked how well the suggested context terms aligned with those they would include manually, 28/40 (72%) of participants responded positively. The terms Scout recommended "*seemed appropriate and helpful*" (P15) and were ones they "*would include in order to narrow the search results*" (P24). Participants specifically mentioned they "*liked having the language already embedded*" (P28) and were "*glad [they] didn't have to repeat it each time*" (P23).

Q1 Summary

Scout significantly reduces the number of searches developers perform, with context terms effectively augmenting developers' searches.

Locating

Participants successfully completed 47% (54/115) of the tasks directly from the information provided in Scout's API signature summaries. In contrast, participants only solved 2% (2/114) of the tasks using just the information presented in the Google results (control). The API signature summaries in Scout provided quick access to the relevant information, and participants expanded an average of two signatures and less than one code example. In nearly half of the cases (24/54), participants also copied information directly from the signature summary into their code.

With Scout, participants opened an average of 0.8 SO posts to access further information, significantly fewer than the 2.2 posts they opened when using the Google results ($t = -6.60, p < .001; d = 0.87$). This 64% reduction in navigation cost can partly be attributed to the information and visual

cues included in the API signature summaries that made it "*easy to find a solution*" (P1, P2, P11, P17, P24, P26, P36, P39). The check mark and most-upvoted icons helped participants "*decide where to click first*" (P15, P39), while the usage and code tabs specifically made locating a solution easier (P6, P17, P25, P23). Participants also liked "*being able to see function signatures of answers*" (P13), allowing them to compare "*different solutions right in the results page..looking there instead of clicking into every link*" (P25). In contrast, the Google results were not always helpful in locating solutions (P9, P16, P17, P24) since the information might be "*not in context and not necessarily very relevant*" (P4). In these cases, participants instead used the title (P4, P39) to "*click into the SO links and read the thread*" (P9) in the full-page view (P4, P10).

Since there are several answers per post that developers might go through to locate a solution, we also examined the number of answers developers looked at. We considered an answer as viewed when at least half of it was visible in the search interface for at least one second based on scroll events. Overall, study participants viewed an average of 2.5 answers per task when using signature summaries, significantly fewer than the 5.4 answers with Google results ($t = -2.82, p = .002; d = 0.37$). When focusing solely on the times a participant opened a post, the number of answers looked at is almost the same in both conditions, with 3.2 answers read per opened post for tasks with Scout and 3.1 answers with Google (no significant difference, $t = 0.19, p = .577; d = .02$).

When opening a post, Scout enabled participants to jump directly to the answer corresponding to a specific API signature via a link, rather than examining each answer starting from the top of a post. Participants used this link for 8 of the 61 (13%) tasks where they opened a post, viewing an average of 1.8 answers once the post was open, including the linked answer.

Q2 Summary

API signature summaries make relevant information directly accessible so that developers open significantly fewer SO posts and look through significantly fewer answers.

Task Completion

Overall, we found that there was no significant difference in participants' success rate or completion times between the two conditions. In terms of

success rate, we marked a task as successful if the participant's solution passed all of the provided unit tests. While participants were slightly more successful when using API signatures, completing 90/115 tasks compared to the 85/114 tasks when using Google results, the difference was not significant (Two-Proportions Z-Test $\chi^2 = 0.2537, df = 1, p = .307$). There was also no significant difference when excluding cases that exceeded the allotted time.

In cases where participants were unsuccessful, we coded the feedback they provided to understand their main challenges under the two conditions. When using Google results (without context terms), participants' main issue was "*finding the right search terms*" (P14; P10, P17, P20, P24, P27). When using Scout's API signatures, "*finding a relevant code sample seemed straightforward*" but participants wasted time resolving variable name typos (P1, P26) and not using the signatures sooner (P25; P33). Despite their similar performance under the two treatments, 36/40 participants indicated a strong preference for API signatures, as P39's feedback illustrates: "*[Google] was missing the organized function [API] call signatures like the previous tasks. I liked those!*".

Regarding completion time, participants were generally able to complete the tasks in the allotted time. We measured participants' completion times from the start of the task to when they clicked done, including the time they spent comprehending the task and implementing a solution (Figure 5.6). While tasks were overall completed slightly faster on average with the control condition (4m18s) than with Scout (4m24s), a two-way repeated measures ANOVA, treating the task as a blocking factor, showed that the treatment had no significant effect on participants' completion times ($F(1, 227) = 0.06, p = .81$). Note that we are missing the completion time for one participant for T6 due to technical issues. In 67 cases, participants exceeded the allotted time, including four large outliers in T1 (19m6s), T2 (27m22s), T3 (17m32s), and T6 (13m32s), with 34 cases in the control condition and 33 with Scout. Ultimately, 6 of the 34 cases in the control condition were successfully completed and 8 of the 33 using Scout. Our results remained consistent when excluding these cases: participants took 3m0s on average for the control condition and 3m6s for Scout, and the ANOVA showed no significant difference.

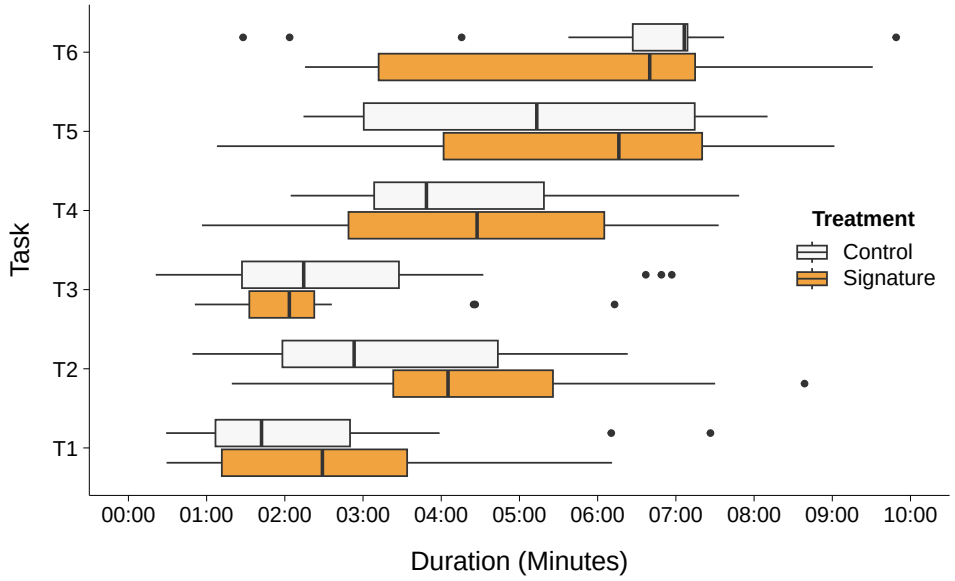


Figure 5.6: Distribution of task completion times. For readability, we do not show the four large outliers.

Q3 Summary

Despite the significant reduction in information when using API signature summaries, there is no significant effect on developers' task success or completion time between conditions.

5.2.5 Tool Performance

It is important that search tools respond quickly with useful suggestions that help developers successfully complete their tasks. Here we report on Scout's signature ranking and processing latency.

Conciseness and Ranking

Overall, Scout provided participants with succinct representations of the information available in their search results by surfacing task-relevant API signatures. The SO posts in the search results for the 69 searches participants made using Scout in T5 and T6 contained an average of 30k words and 292 LoC with 122 signatures across 26 answers. Instead of participants

manually opening posts and looking through the answers for the appropriate API information, Scout provided a concise overview, summarizing the results into 23 signatures using 86 words and 23 LoC on average.

To examine how effectively Scout ranks API signatures within each SO post, we manually compared participants' solutions with the API signatures shown by Scout for the last search made by each participant during T5 and T6. For these searches, there were 15 signatures on average in each post from which Scout selected the 3 most relevant considering the participant's code. We found that 31% of participants' solutions used the first signature, 64% used one of the first 3 signatures, and 97% used one of the first 10. For some tasks, participants were impressed with the signature ranking noting that "*the first item..had the answer I needed*" (P32; P20). However, the ranking was not always perfect (P9, P28, P29, P31, P35) leading participants to examine signatures that were not always relevant (P5, P10, P32). In these cases, signatures can help mitigate the impact of ranking by enabling developers to directly assess the relevance of the results without opening the full SO post: "*It was really easy to find the answer, even though it wasn't the first one because [the signatures] were nicely collapsed. The example also helped*" (P31). Participants also suggested that the ranking could be improved by incorporating the number of upvotes (P27) and answer recency (P8, P24) into the ranking algorithm, and prioritizing signatures with shorter code examples (P26, P32), or those meeting technical requirements such as the language revision (P20).

Latency

Scout creates the API signature summaries by processing the first 10 SO posts returned within Google's search results. Since this processing happens in real time and can impact Scout's usability, we analyzed the latency it adds to the Google search results. We found that Scout took an average of 4.4s to display the first search result and an average of 5.0s for Google's top-ranked result, on the 8-core virtual machines used during the study. While this latency is below the 10s tolerable wait time for web-based information retrieval tasks, it is longer than the sub-2s ideal wait time [97]. According to participants, the latency was not a major factor affecting Scout's usability and is partly offset by reducing navigational overhead (Section 5.2.4), which may result in an overall faster "*experience*" (P15). However, some participants did mention that the "*search could be faster*" compared to Google's highly-optimized search engine (P5, P8, P11, P18).

5.2.6 User Experience

For developers to actually find solutions using a search tool, it must be usable and support the types of searches developers make during their tasks. We examine these aspects below.

Usability

Participants indicated that Scout has *good* usability, assigning it an average system usability score of 78 (a score of 70 is average) [10]. They found it "*helpful to have [the results] in the IDE*" (P39) so they did not "*have to leave the [..] code*" (P15) and "*loved that [they] didn't have to switch tabs/windows*" (P19). However, one usability challenge was the limited space available in the IDE (P4, P6, P8, P11, P16, P18, P23, P29, P37) which made the interface "*a bit too busy*" (P28) and "*slightly harder to navigate*" (P13).

When comparing the API signature summaries with standard Google result summaries, participants generally found that signatures were quicker to scan for useful solutions saving them time and mental energy processing full SO posts. However, participants noted that with signatures it could be "*a little tough to know which result to use [from] the initial view*" (P15) without more documentation (P5, P6, P13, P23). In contrast, the textual content of standard Google results resized to fit the space available in the IDE but required additional navigation to "*dig*" through code, comments, and links to get an actual solution (P20, P23).

Search Suitability

Participants indicated that signature summaries are best suited to searches involving *basic* tasks where they are familiar with the overall approach and need to *recall* details about *common* API calls such as syntax using *code examples*. However, they are not as useful for searches involving *specific* or *complex* tasks that require a *detailed understanding* of possible solutions using *multi-line code examples*, or for tasks that are *conceptual* in nature. Participants reported that 58% of their day-to-day searches are for basic tasks while 38% are for complex tasks. To better support these two types of searches, participants suggested a hybrid approach "*having both [summaries] at once*" (P15, P31) by adding full API signatures or even "*just function names*" (P20, P23) to the Google results.

IDE Integration

Participants were generally positive about Scout's integration into the IDE, but also suggested some ideas to further the integration. Concretely, participants suggested adding a "*copy*" button (P28, P33) that "*drops the [summary] code into the source file*" (23), and altering the summary code to match the source code; e.g., by "*matching the parameter [names]*" (P31) and "*highlighting variables*" (P16). To maximize the limited window space, participants suggested using "*[line wrapping] for signature summaries to reduce horizontal scrolling*" (P24), and providing "*tooltips to view text snippets related to code*" (P4, P24) with an option to "*expand the Scout window or open the links in new windows*" (P18).

Q4 Summary

Participants valued Scout's concise API signature presentation and the tight IDE integration that speeds up scanning and locating solutions, but noted that the amount and presentation of the information could be improved. Scout is particularly suitable for basic search tasks comprising more than half of developers' day-to-day work.

5.2.7 Limitations

We describe the limitations of our study below.

Internal Validity This study uses a within-subject design which introduces confounds such as carryover and ordering effects. We randomly assigned participants to one of four trial groups, following a balanced Latin Square design, to help mitigate these effects, and included breaks to limit participant fatigue.

Before starting the study, we conducted a power analysis to estimate the number of participants necessary to observe statistically significant results. While we used the recommended effect size of $d = 0.5$, our post-hoc power analysis indicates the actual effect size was much smaller at $d = 0.13$ limiting our interpretation of success rate and completion time. By using many shorter tasks, participants' overall completion times may have been dominated by aspects of the task other than searching, including reading instructions, comprehending code, and testing solutions.

External Validity The results from this study may not generalize to all developers and tasks. We chose to run the study primarily on paid platforms to recruit a large number of diverse participants who would have been difficult to contact directly. However, by conducting this study on paid platforms, participants may have been incentivized to misrepresent their ability or otherwise complete the study differently than expected (e.g., by working on multiple studies simultaneously). We used a validated questionnaire to prescreen participants, and required participants to successfully complete the tutorial task and attempt all of the tasks before being compensated.

The tasks we used for the study were designed to represent individual components of larger features found in real software systems. We chose to use short tasks, presented in isolation, to provide more comparison points and to help ensure participants could make progress while keeping the overall study duration reasonable. We intended this design to match how developers decompose their tasks and search for solutions in practice.

5.3 Discussion and Summary

We conclude this chapter by describing how Scout contributes to prior efforts and discuss some challenges and solutions for extending Scout to developers' workflows in practice.

Summarizing Task-Centric Results

Text snippets, like the ones typically used by web search engines, can be generated robustly but have inherent limitations when summarizing developer-specific results such as being "*out of context*" (P4), containing "*unnecessary details*" (P17), and preventing developers from "*easily seeing and comparing answers*" (P15). For searches about APIs, we found that signatures address these limitations by enabling developers to locate solutions for their tasks more directly. To make the summaries *easier to navigate*, participants suggested including the age (P10), number of votes, and source answers for each signature (P13, P24), and always providing the full signature "*showing what parameters the function accepts, rather than just based on its usage in that specific answer*" (P38). Participants also suggested removing the Usage tab (P23), or defaulting to the Code tab (P29) to "*save an extra click*" (P19). For code examples, participants wanted "*longer documentation*" or a "*small summary*" (P5, P36) with "*comments in the code*" (P39, P7) and links to the function's official documentation (P10, P13, P36).

While signatures can help developers save time by directly surfacing the API information contained in their search results, the latency from processing the search results in real time can diminish Scout's overall time savings. The primary factors contributing to Scout's latency are fetching the posts from SO and processing the code examples into ASTs. In future work, we will examine approaches to minimize real time processing latency, for example, by requesting posts directly through the SO API rather than using proxied network requests, and by caching the code example ASTs.

Completing Tasks

We designed Scout to reduce cognitive load and help developers focus on their tasks. However, it is difficult to reliably measure these aspects directly so we instead examined completion times and success rates, which are common measures when comparing alternative tool designs. We did not find significant differences in completion time or success rate between the Google and Scout conditions. On one hand, we view this as a positive result since we compared Scout's novel interface, which participants had to learn while completing actual development tasks, with Google, one of the most commonly used web search tool by developers. On the other hand, we expect Scout could be faster in practice by enabling developers to search directly in the IDE, avoiding the cognitive overhead associated with the context switches and window management that occur when using a separate browser application. For this study, we compared both conditions within the IDE to avoid any confounds and privacy concerns outside of our main evaluation of Scout's API signatures. In future work, we will evaluate Scout in the field to determine if it actually improves the time and success rate for developers to locate API information during their actual web search tasks.

5.3.1 Contextualizing Developers' Searches

Constructing effective queries can be difficult, leading developers to invest time and mental effort examining results which are not relevant to their task. We found that many of the terms developers include in their queries to filter the search results can be automatically extracted from their source code and, by presenting the terms explicitly, enable developers to actively guide the search process. Participants found the terms that Scout included in their searches aligned with their expectations and even thought that additional terms, such as variable types, and terms from specially formatted

comments like `JSDoc` (P4, P26, P28, P37), would be helpful. However, they acknowledged that it would not always "*reasonable [..] when [the terms] don't show up in code*" (P31). Some participants were concerned that Scout might "*infer too [many terms]*" (P15) from longer functions that could introduce noise into their searches (P8). Future approaches could automatically include the 2–3 terms closest to the developer's cursor and offer remaining terms through autocomplete. Participants also thought that providing "*suggestions like Google's 'Did you mean...?'*" (P19) could help them "*search for similar problems*" (P2).

5.3.2 Alternative API Oracles

The focus of our experiment was to study the effect of using API signatures to orient developers within their search results and help them locate a solution more directly, independent of where the API signature information was derived from. While there are existing approaches that share similar goals, such as Prompter [111], PostFinder [124], and Biker [56], they focus on improving the underlying search results through custom search engines. Instead, Scout focuses on aligning the presentation of the information contained within a set of search results to the developer's task. By decoupling the presentation of search results, Scout works on top of any search engine providing SO results. Due to its ubiquity, quality of results, and ease of use, we chose to use Google as our baseline search provider within Scout.

Tools based on LLMs, such as ChatGPT and Copilot, are another source of API information but, similar to web search, require developers to either craft effective queries or look through the responses to identify relevant information. In the case of ChatGPT's conversational interface, developers often have to try multiple queries to obtain a complete solution for their task [98]. This process adds overhead as developers have to read the frequently verbose, and structurally variable, responses and think about how to refine their queries to elicit the desired information [62]. Copilot avoids queries by generating multiple code solutions from cues in the developer's source code. However, the code-only representation of solutions makes them difficult for developers to fully assess, which is important since the solutions can be wrong in subtle ways [100]. Further, the size of the solutions, and Copilot's autocomplete interface, which provides one solution at a time, can make it hard for developers to efficiently compare alternative solutions. The presentation system used by Scout can be applied to existing LLM tools to enable their recommendations to be presented in a way that is better tailored to the developer's task.

5.3.3 Prior Efforts

Developers perform two steps to locate solutions on the web: first, they generate search terms and then they navigate among the result pages. Prior work has generally approached these steps independently by creating tools that help developers either construct search queries using code terms in the IDE or navigate the result pages by transforming their content using different summarization techniques in the web browser.

Constructing Search Queries

Coming up with effective search terms is difficult and often requires developers to refine their terms after investigating the results, which can take considerable time [114, 154]. One way to minimize this refinement phase is to use keywords from the text in the IDE, such as source code and error messages, to generate queries automatically. An early example of this approach is `Strathcona`, which automatically recommends API usage examples from an oracle based on the structural context of the source code selected by a developer in their IDE [53]. `Pycee` and `Maestro` use stack traces to generate searches to help developers locate information about program errors [83, 141]. `Fishtail`, `Reverb`, and `Prompter` use the code entities developers have recently interacted with to recommend previously viewed web pages ([128, 129]) and relevant SO posts ([111]). `StackInTheFlow`, `PostFinder`, `FaCoY`, and `Bing Developer Assistant` generate searches on-demand from the code selected by developers in the IDE [45, 69, 124, 153].

However, these approaches rely on custom search indexes that may not always have the most up-to-date information (e.g., SO data dumps). It can also be difficult for developers to guide these approaches using their knowledge of the task since the search queries are generated and run transparently. Instead, Scout offers code terms that developers can use to augment their queries, similar to `Blueprint` [24], allowing them to actively control the results from standard search engines with minimal effort.

Transforming Search Results

Generating effective queries is only the first part of locating solutions on the web: developers still need to find the specific kinds of information relevant to their task from their search results, such as explanations and examples of API usage [54]. Unfortunately, finding APIs within the search results can be difficult when developers are only given limited information about the result such as the title (e.g., `Google`, `Prompter`, `PostFinder`).

One way to help developers effectively navigate the results is to annotate them with developer-specific information. `Mica` and `Assieme` extract the function and type names, respectively, from search result pages so that developers can easily assess and refine the results [52, 138]. `Libra` adds a plot to Google's search result page visualizing results according to the number and diversity of overlapping code terms, helping guide developers toward pages with more details or alternative solutions [112]. A tool by Liu et al. identifies latent developer needs within SO questions to help developers identify posts relevant to their task [80].

Alternatively, approaches can provide a summary of the results so developers do not need to examine each result individually. `Example Overflow` presents the top 5 accepted code examples from SO search results in a single view allowing developers to easily compare them [152]. While such code examples can be helpful, they can be time-consuming to read and may not reflect common usage. `Exempla Gratis` identifies common usage patterns across multiple code examples to identify the idiomatic usage of developer-specified API methods [15]. Instead of code examples, `AnswerBot` focuses on textual content to extract a diverse summary across answers to give developers an overview of the entire SO post [150].

Some of these approaches have focused on summarizing API information. `CAPS` summaries SO posts describing API issues to help developers improve the API design [4], while a tool by Campos et al. identifies highly-voted answers across SO posts demonstrating how to use an API [30]. `Opiner` provides sentiment about Java packages found in SO code examples based on the answer text to help developers decide between alternative APIs. `Biker` uses heuristics to extract API entities from hyperlinks and code tags in the top-50 SO posts related to a developer's search by comparing them to a curated oracle of Java API names. The entities are mapped to their fully qualified name and ranked by frequency and similarity of the post title with the search terms. The API description, a list of similar questions, and code examples are presented in a dedicated browser window using plain text [56].

While we share a similar objective with `Biker`, our approach differs in several key ways. We focus on presenting APIs interactively in the IDE where developers can compare different options before expanding detailed usage examples. Our approach works in real time using Google search results avoiding the need to construct and update a custom search engine. Finally, we consider where the extracted APIs will be used in the developer's code when making recommendations.

5.3.4 Summary

We designed Scout to help developers locate API information within their search results by extracting, ranking, and presenting compact API signatures directly within the IDE. Scout works on top of existing search engines by incorporating terms found within the developer's source code to augment their search queries and adjust the search results. We found that presenting search results as API signatures enabled developers to easily compare different APIs and interactively drill down to details when needed, significantly reducing the amount of information they examined. Participants were able to complete API search tasks with Scout's novel interface while maintaining similar task completion times and success rates.

Our findings indicate that the way information about source code is presented can affect the amount of effort required for developers to complete their tasks. Code context provides an effective mechanism for tailoring the representation of developer's information resources to their task. In the case of API search tasks, an API signature representation helps developers assess the relevance of search result pages and, for many searches, provides exactly the required information.

RQ3 Summary

The information about the types used in the developer's active source code, together with the programming language and library names provide context that enables Scout to tailor Stack Overflow search results to the essential API information required by the developer. An API signature presentation helps developers to locate solutions more directly and enables web searches for API information to be integrated into the IDE. This integration mitigates integration friction associated with conventional API search where developers must switch and transfer information between their web browser and IDE.

Chapter 6

Towards a Proactive Workspace

Friction arises from the tool-centric design of the workspace: developers work across tools to complete their tasks but, as tools are independent from each other, developers have to perform extra work to coordinate the state of their tools to align with their tasks. For example, developers have to set the right working location across tools, navigate the file system to open task resources, and close windows that are no longer relevant to their task (Chapter 3). While setting the state once for a single task within an individual tool is generally quick, developer's diverse tools and frequent task switching compound the impact of friction on developer productivity.

To help developers focus on their tasks, rather than the mechanical aspects of coordinating their tools, we examined whether sharing context information between tools could help mitigate friction. We evaluated the effect of automatically sharing different kinds of information between tools through two prototype approaches. Helm observes the locations of the resources developers access. By incorporating this information and the active project, Helm was able to recommend and help developers re-visit resources more directly (Chapter 4). Scout uses terms from the developer's active source code, including the programming language and library names, to help surface API information when developers search online. The improved search results, together with Scout's API signature presentation, reduced the amount of navigation necessary for developers to find a solution (Chapter 5). These findings provide evidence for our thesis statement that collecting and using context from the resources developers access across their tools can reduce the friction developers experience locating information.

In this chapter, we synthesize the results of these two approaches to formulate a vision for *proactive workspaces*. The core idea behind proactive workspaces is that, by sharing lightweight context information, tools can proactively adapt to the developer's task and reduce friction.

6.1 Why be Proactive?

Proactive workspaces would directly address challenges we have observed developers facing while working on their tasks in practice (Section 3.1.1). An exemplar workflow for a development task performed by S1 who was addressing a table sorting bug is shown in Figure 6.1. Despite much of the setup steps already being complete (pulling changes, creating a branch, and locating the bug) the workflow still required developers to manually navigate and coordinate between between multiple independent tools. While the coordination steps were not difficult, they all required manual effort that took the developer away from the core task they were working to accomplish.

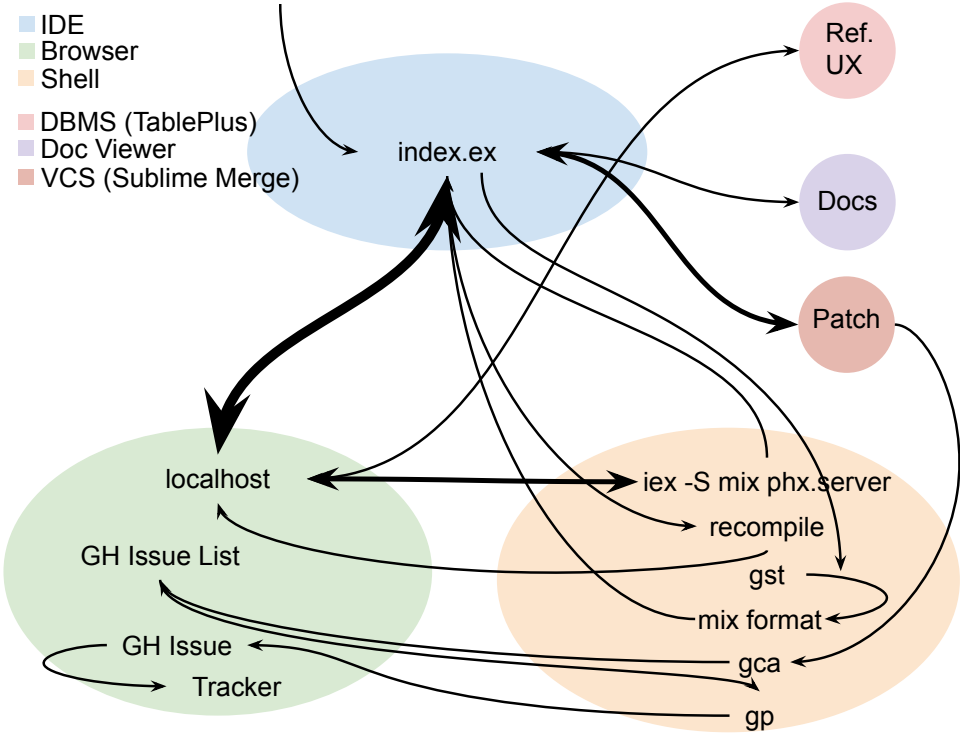


Figure 6.1: Actual bug fixing workflow used by S1. Edges represent switches between tools with thicker edges representing more frequent switches.

6.1.1 Friction-Induced Unnecessary Work

Development friction can capture any form of interference a developer encounters while trying to accomplish useful work. Usually, friction manifests

as gaps between the developer's task and how directly their task is supported within their workspace. Each project and developer has their own curated set of tools. Current workspaces require developers to manually coordinate all of the tools necessary for their tasks. This often involves switching between tools, complex navigation within tools to locate task-relevant information, and extensive tool customization as they work. Differences between tools makes navigation and configuration a tedious manual process that distract developers as they try to focus on the essential aspects of their tasks.

Concurrently, variations in the frequency, complexity, and duration of tasks means developers often switch between tasks. Tools generally only support one task at a time, forcing developers to reconfigure each tool when they switch tasks, further increasing coordination overhead. Task information is often encoded in the *state* of the tool, which includes the set of open files, execution output, and action history. When re-configuring tools for a new (or prior) task, some of this information is lost and other information can be challenging or laborious for developers to recover.

6.1.2 How do Frictions Manifest in Practice?

Here we describe the work that one developer, Alex, might complete in a 90 minute development session using a current workspace. This example is a synthesis from the live-streamed developer videos we analyzed, and illustrates how current workspaces require developers to manually coordinate their tools through a series of navigation and configuration steps before they can actually work on their tasks. While this example may seem simple, it captures how existing workspaces force developers to manually orchestrate and perform many low-level actions to accomplish their work; this task is in no-means unique: developers encounter this kind of friction dozens or hundreds of times each day.

Collecting information. Alex wants to get their next task from their team's bug tracker so they open their web browser, type 'github' into the address bar, and select the first autocomplete suggestion to load the GitHub bug tracker tool. They spend a confused minute scrolling through the bug descriptions until they realize that the autocomplete opened the tracker for the wrong project. After navigating between projects to the correct bug report within GitHub, Alex starts their IDE to work on a fix. Since Alex's coworker said that they added a test for the bug, Alex started by running all the project's tests, but found they were all passing. Alex went back to the issue to recall the exact test name and, after switching back-and-forth a

few times, confirmed the test was missing. After thinking for a minute, Alex realizes that they forgot to pull the latest changes, including their coworker's tests. These steps required Alex to open and navigate in the browser (to read the bug report), navigate in the IDE (to find the tests), and perform actions (both version control and test running) in the terminal.

Handling interruptions. Alex decides to take a break from fixing the bug to think about possible solutions. Realizing that they have been putting off a large code review, they open a new tab in their browser and navigate to the pull request (PR). Using GitHub's diff view, they start reading the first few changes. However, the changes reference other files and Alex realizes that they need more context from the code surrounding the change. Alex decides that it will be easier to simply check out the PR and read it in their IDE. Switching to their terminal, Alex manages to navigate to the project directory after several rounds of `ls` and `cd`, and checkouts the branch for the PR. After repeatedly switching between the code files in their IDE, the PR description, and comments explaining the changes in another browser tab, Alex understands what their coworker was trying to accomplish and has some suggestions. Alex goes back to the GitHub diff view in their browser and manually locates each of the problems they saw earlier in the IDE to submit their comments.

Finish tasks. Alex is now ready to resume fixing the initial bug but they want to re-read the bug description to make sure they are not overlooking anything. After switching tabs a few times, Alex becomes frustrated that they cannot find the bug description among all their open tabs. They close all their browser windows to clear the tabs, not worrying about all the information they had open that might be helpful in answering follow-up questions in the code review, and then re-navigate back to the bug description. Satisfied that their solution addresses the bug, Alex switches back to their terminal to commit their changes but realizes they are on the wrong branch. Alex initially tries to look up the branch name using the history feature of the terminal but is discouraged by the amount of unrelated commands that were captured from other prior activities, including the routine navigation commands, `ls` and `cd`, they used for locating the PR. Instead, Alex switches to GitHub and navigates to the branch list to find the name. Finally, Alex creates a commit, referencing the issue number in the commit message, and pushes their changes for review. In all three of these sub-tasks, Alex had to configure each of their tools to ensure the resources were avail-

able to help them perform the essential work required for the task. While these seem trivial, each of these manual steps was a form of accidental complexity where the developer had to manually perform work that needed to be done, but was not essential, for their task due the independent design of tools in current workspaces.

6.2 Addressing Friction with Proactive Workspaces

One of the most common frictions developers experience is locating information across dispersed resources. Our two approaches, Helm and Scout, address this friction through two main concepts: capturing relevant context information across tools and using the context to recommend and tailor resources for the developer's task. However, we implemented Helm and Scout as independent tools. While appropriate for investigating the underlying idea of sharing context between tools, new and independent tools can add further friction to developer's workflows. Our vision is that the workspace could provide workflow continuity for developers by automatically capturing and sharing context to enable tools to proactively align with the developer's task.

In current workspaces, every time a developer enters a tool, they must configure it for their current task which often involves changing the tool's state. A proactive workspace would automatically adapt a tool to the developer's task based on the actions they were performing in their *last* tool; in this way developers can move seamlessly between tools as they work. These proactive actions would be especially useful during developers' frequent task and tool switching to naturally shift the state of each tool to the current task without imposing any changes to the way the developer works. Similar to how autocomplete frees developer's from manually identifying valid code elements, providing tools with automatic customization support would significantly ease the unnecessary work developers commonly perform when identifying and performing the tedious actions necessary to prepare a tool for the current task.

A proactive workspace would capture information about the content developers are working with and provide it as context to tools when the developer activates them. Tools would use this contextual information, along with the tool's own understanding of how the developer has interacted with the tool in the past, to tailor itself for the current task. The workspace would provide an environment that runs context-dependent actions when the de-

veloper switches tools. Tools would register their own actions for handling the context in a way most appropriate for the tool, e.g., by tailoring the set of open files, changing into different working/project directories, filtering application histories and communication channels, managing the state of the version control system, and setting environment variables appropriate for the developer's current task. Tools could either tailor themselves automatically or show a simple dialog when they are activated to enable the developer to quickly review these suggested actions and either accept them, updating the state of the tool, or dismiss them leaving the tool in its previous state.

A key observation from the development sessions in Chapter 3 is that, while developers have unique ways of working, the structure of their workflows at an individual level are highly regular across a limited number of tools including continuous integration systems, code review tools, IDEs, terminals, and various code-related websites. Coordinating the state of all these tools as developers switch between their tasks often requires developers to manually perform many routine steps, not because the facilities to automate the steps are absent, but because the tools do not have access to the context necessary to parameterize the steps. The dominant design philosophy of existing tools is that they are independent: individual web domains (e.g., GitHub and StackOverflow) do not interact with one another; they also do not interact with the files open in the developer's IDE, or the text output being shown in the terminal. While existing tools have design independence, they do not have context-independence: the files in GitHub correspond to files in the developer's IDE; the code snippet in a StackOverflow answer can correspond to code in the an IDE tab; and the error message in the shell can correspond to a tutorial online.

6.2.1 Concrete Behaviours that Address Friction

Reflecting on the design of our Helm and Scout tools, and synthesizing the feedback developers provided about how the tools affected their workflows, we describe concrete examples of proactive behaviours that would help mitigate friction in developers' workspaces. With these examples, we demonstrate how proactive workspaces, while representing a meaningful shift in the complete independence of these tools, could assist developers with some of their most common steps by exposing only a few pieces of contextual information. Table 6.1 links the workspace design aspects that induce friction (Section 3.4) with aspects from our Helm and Scout approaches that could help workspaces be more proactive.

Table 6.1: Mapping aspects of the Helm and Scout approaches that we evaluated to address workspace frictions.

Friction	Evaluation	Proactive Behaviour
Access	Helm: re-establish location	Set Working Location
Access	Helm: cross-tool project history	Partition Tool Histories
Integration	Helm: resource preview Scout: tailor content to fit IDE, augment search terms	Link Resources Across Tools
Translation	Helm: script builder	(Section 6.3)

Helm Retrospection With Helm, we examined whether access friction could be addressed by providing a centralized interface that enables developers to navigate directly between resources without needing to think about whether the resource is already open and in which application and window it is located. Helm enables developers to automatically re-open resources across applications to the correct location for their task. In some cases, developers can avoid navigating to resources by viewing them directly within Helm, where they can, for example, copy information without leaving their current resource. For the shell in particular, Helm automatically applied the correct working location for the commands developers selected to re-execute.

However, by enabling developers to interact with resources from across different tools, Helm re-implements limited versions of existing tools. While carefully selected features can improve productivity, too many features can become a hindrance to usability, a concern shared with modern IDEs [21].

Scout Retrospection Scout uses a progressive disclosure design to tailor web search results around API signatures allowing developers to investigate promising solutions without navigating through unrelated information. The compact API signature representation enables search results to be integrated into the IDE, adjacent to the developer's code where the API information is needed. By surfacing API information directly, developers avoided switching between different applications and were able to locate, compare, and transfer solutions into their code more easily. However, the limited space in the IDE means that direct integration is not always an ideal solution. Linking code and web resources through a shared context would enable the browser to

tailor the search results without space constraints. By keeping the web and code resources visible simultaneously (e.g., [3, 60]), developers would still be able to easily compare and transfer solutions.

Set Working Location

Tools typically operate with a particular working location often tied to a specific project such as a directory on the developer's local filesystem or the base path of a URL. Developers frequently have to (re)navigate to their project location within their tools. Friction arises due to disorientation resulting from the misalignment of tool locations relative to the developer's task, and the significantly different structures developers must navigate to set the correct location (e.g., the local filesystem, GitHub URL, AWS deployment URL). By maintaining project identifiers, which associate the locations of each project across all of the developer's tools, proactive workspaces could automatically set each tool's working location. This identifier could be set by the developer the first time they begin working on a project or inferred from the developer's navigation. Based on our observations, proactive navigation would be especially useful in two of the most commonly-used tools, terminals and browsers, whose standardized interfaces enable proactive workspaces to perform the necessary actions.

Partition Tool Histories

Tools often maintain histories of the resources developers have previously used. When developers need to re-find a resource, they often begin by examining the history, but this is not always successful. One challenge is that, while developers are typically thinking about their resources in terms of their projects, tool histories are often organized by time, resulting in resources from different projects being intertwined. This discrepancy between the developer's needs and how the history is organized, can cause developers to become distracted (e.g., by opening the wrong resource) and resort to using alternative search strategies (e.g., manually navigating individual resources). The active project identifier maintained by the proactive workspace would enable tools to partition their histories based on the developer's task.

Another challenge developers encounter with tool-specific histories is simply remembering which tool they used to view the resource (e.g., when using multiple web browsers). Beyond providing task-centric partitioning to help developers locate resources within a specific tool, proactive workspaces could further unify these disparate histories enabling developers to re-find

resources from a central location.

Link Resources Across Tools

Developers often need to work with the same resources in multiple tools as they progress through their task (e.g., finding the same file in the tools they use for editing, version control, code review, CI/CD). However, as information about the location of their resources is siloed within the individual tools, developers have to manually locate each required resource. Proactive workspaces could assist developers transfer resources between tools by automatically resolving incomplete locations (e.g., containing only file names) and mapping between local and web-based locations.

In addition to transferring resources between tools, developers also often use the content from their last viewed resource to identify related resources. By observing the content that developers use across tools, including where it originates from, and where it is used, proactive workspaces could automatically capture the content and make it available for tools to make recommendations.

6.2.2 Improving the Motivating Workflow with Proactive Workspaces

Using the workflow from Section 6.1.2, we describe how a proactive workspace could have helped Alex avoid unnecessary work and enabled him to more directly focus on the essential parts of his task. Initially, Alex encountered difficulty starting his task because many of his tools were configured for what he was working on previously (e.g., a previous project in GitHub's bug tracker). However, Alex's interactions with other parts of the workspace provide strong cues about his intentions. For example, when looking at the issue assigned to him, Alex has implicitly identified the project for his current work item, as the bug tracker is typically associated with only a single project. By surfacing the project identifier to the proactive workspace, any subsequently used tool would then be able to know which project it should activate. For example, when Alex switched to the IDE, it could have proactively *set the working location* rather than restoring the last opened location.

Using contextual information from Alex's initial examination of the bug report, the IDE could have also suggested a short set of files to open. Most notably, the IDE could identify that a specific test case was of interest when Alex was examining the bug report, and that the file containing this test

case should be the active tab with the test case scrolled to be visible on the screen.

Midway through the task, Alex realized he needed to perform a code review. When he opened the code review tool in his browser, the workspace was notified of the project related to the change. This meant that when Alex switched back to the IDE to look at the change in more depth, the IDE was able to change to the correct project, and open the *files linked from the code review tool*.

Finally, when Alex was ready to submit his bug fix and attempted to use the terminal history to checkout the correct branch, it had been *partitioned by the project* that was set after Alex revisited the issue. The filtered history showed the command to checkout the required branch as the first entry.

Impact on Unnecessary Work While each of the steps the proactive workspace helped Alex with in the prior example are small, together they represent a large number of manual steps in the terminal, web browser, and IDE that Alex would have otherwise had to perform himself. The most notable benefit of the automation enabled by tools having knowledge about what Alex was recently doing, though, is that Alex was able to keep thinking about his *task*, not the operationalization of the manual steps he needed to perform when switching tools.

6.3 Automating Workflows

During our investigation into how developers structure their work (Chapter 3), we observed that developers often had to (a) manually split their high-level tasks into low-level actions that are supported by their tools, and (b) (re)establish their current context in each tool. We found that these low-level actions created *translation friction* by requiring developers to plan out the workflows necessary to achieve their tasks.

In prior work, we examined a Conversational Developer Assistant (CDA), called Devy, that automatically performs the workflows developer's request through a voice interface. Devy reduces the number of manual, often complex, low-level actions that developers need to perform which helps address translation friction by freeing developers to focus on their high-level development tasks. Specifically, Devy infers high-level intent from developer's voice commands and combines this with an automatically-generated context model to determine appropriate workflows for invoking low-level tool actions. The automatic mapping and execution of workflows

based on the developer's high-level intent, augmented by the context model, reduces developers' cognitive effort of breaking down high-level intents into low-level actions, switching context between disparate tools, and parameterizing complex workflows.

In contrast to proactive workspaces, which aim to keep tools aligned with the developer's current task, Devy's automation goes further and reduces the number of tools developers need to interact with to complete their tasks. While Devy's automation reduces translation friction, we found that it introduces trade-offs in terms of task specificity, the number and diversity of tasks developers can complete, and transparency of what the automation is actually doing. Proactive workspaces are meant to bridge these trade-offs by offloading unnecessary work from developers while allowing developers to work using their existing tools and workflows.

A key aspect of the Devy approach is centralized context model that helps promote context to a first-class construct in the workspace [95]. Rather than each individual tool hooking into other tools directly to collect information (e.g., as we did with Helm and Scout), a centralized context facilitates tools sharing contextual information with each other.

6.3.1 Concrete Information Sharing Mechanisms

Proactive workspaces enable tools to share information they need to align with the developer's task. Uniform resource identifiers (URIs) provide a structured and well-established mechanism for exchanging information. In proactive workspaces, when a developer switches tools, the last tool provides the next tool with a URI pointing to the resource the developer accessed before switching. Upon activation, the next tool would use the URI to get the information from the resource the developer accessed previously. The tool would use the information, under its own mandate, to better align itself with the developer's task.

One downside of tools directly exchanging information is that historical information about the developer's activity is not recorded. A broker service would facilitate the exchange of information by receiving and *recording* the URIs of the resources developers access as well as forwarding URIs to the appropriate tool where the URI can be dereferenced to its corresponding content.

Capturing URIs. Applications would expose the URI of the resource that was active when the developer switched away from the application. The broker service would pull this URI when the application loses focus and

make it available to application gaining focus. Exposing the URI in this way would be similar to mechanisms already in place in existing workspaces which expose the application name and title.

In addition to a pull-based mechanism, applications could also push URIs to the broker service by hooking into their existing events. For example, applications could push the URI each time a developer activates a resource within the application.

Using the same push-based mechanism, tools within the application could provide even more granular information about resources developers access. For example, while the web browser pushes the URL of each website the developer visits to the broker service, developer-specific website such as Stack Overflow may push additional details about the specific content the developer focused on (e.g., answer post).

Ultimately, each application and tool decides whether to share URIs, under what circumstances, and at what granularity enabling flexible and incremental adoption.

Dereferencing URIs. When the active tool recognizes a URI it supports, the tool requests the URI contents from the broker. The broker does not store or supply the content but instead forwards the request to the tool that originally provided the URI. The original tool would use its own URI to recall and supply the content that the developer had previously accessed. The requesting tool, broker service, and source tool would negotiate the format of the content similar to how RESTful services and clients negotiate content formats.

URI Relationships and Developer Activity. Content is only one form of information tools can use as context. The duration and order in which developers access resources also provides information about relationships between resources and the task [42]. In addition to requesting the broker service to dereference URIs, tools can also pose queries to the broker service about the relationships between URIs. For example, a tool could request all the URIs the developer has accessed immediately before or after the current URI to gain a better understanding of the developer's task.

Extensibility. Developers can extend the broker service by writing scripts that execute in response to certain URIs and content. For example, developers could add scripts to further customize the behaviour of their tools or remove extraneous content before passing the content to the receiving tool.

6.4 Future Research Directions

The dispersed resources and independent design of current applications does not support the way developers actually work; proactive workspaces could align with developer needs much better and reduce the unnecessary work they are forced to perform. In their reflection paper, LaToza and Myers argue that the research community should design tools that solve problems developers actually encounter considering the problem's frequency, duration, and impact on code quality [73]. Similarly, for generative AI, Ozkaya calls on the research community to ask what barriers these tools remove and how they can be incorporated effectively into developer workflows [104]. We have observed that tool coordination is an almost constant problem which creates task overhead, reduces tool mastery, and requires extensive configuration and maintenance effort from developers, reducing their productivity. We conclude this thesis with three avenues of future work necessary to address friction by enabling the full proactive workspace vision.

6.4.1 Context in Developer Workspaces

Proactive workspaces provide mechanisms which enable tools to exchange information. However, tools still need to decide which information they will expose and which information they will consume. While individual tools will ultimately decide how they will use information to support developers, it may help tool designers to understand how their tool, and the information it exposes and consumes, fits holistically within the developer's workflow.

As a first step towards this understanding, we describe the information that we observed developers using between Continuous Integration systems, code review tools, IDEs, the terminal, and various code-related websites in their workflows (Section 3.1.1). However, new research into the ways developers use information between tools, the impact of the information on the way developers use the tool, and how the information should be shared, remain important open problems in this space.

Copy/Pasted Content. A key observation from developer workflows and prior work (e.g., [5, 54]) is that developers often use terms taken from the textual content displayed in different applications to parameterize tools and integrate information between resources. Table 6.2 provides an initial set of terms developers frequently copy and paste.

While terms are often easy for developers to select and copy, accessing terms programmatically requires parsing and identifying the resource con-

tent into a consistent representation as we indicate in the *Source* column.

Term	Description	Source
Resource paths	Location of the active resource	Provided by OS
Project references	Project names and locations	Derived from path and title of active resource
Task references	Bug/feature report IDs	GitHub issue URL
Branch names	Names of branches used by developer	`git branch` output GitHub URL
Project tools	Shell commands related to the project	Shell command code elements
Environment vars	Name and value of environment variables	Pattern match
Language	Name of the programming language	File extension
Library	Name and version of libraries and frameworks	Program AST

Table 6.2: Examples of development terms developers frequently copy between applications.

Changes in Tool Output. Developers are constantly modifying their software to add or improve features. As they are working, developers will frequently run and re-run their tools to check the status of their code, and will respond to problems by locating, applying, and testing solutions in cycles [121]. When problems with the code are detected, e.g., a test failure or compilation error, the output of the tool will be different from the previous execution. This output may prompt the developer to update their code after seeking out a solution. The output when running the tool again would indicate whether the developer has solved the problem. Despite its potential use for identifying what problems developers are working to solve, and when the problem has been solved, developers have to manually recall the output of consecutive tool runs to identify their progress as each tool run is treated independently.

History Logs. Most applications record information about the resources developer access which help developers re-find information. Developer-specific applications such as the IDE also record the actions developers perform including the tools that are run and edits made to the source code so developers can undo their actions. However, there are limitations in the way applications provide the history to developers. The steps developers perform within an application are often recorded as low-level actions which are too granular to represent the developer's high-level goal. At the same time, application histories only capture entire resources which may be too broad to represent the specific content developers need to revisit, and it can be difficult for developers to associate these histories across applications. Despite these limitations, application histories provide information that helps developers complete their tasks.

6.4.2 Task Identification

A fundamental component of proactive workspaces is the idea of a task, which links context with the developer's actions across tools. While previous approaches, which require developers to manually specify their task, have provided a limited form of proactivity by helping developers organize their resources (e.g., [37, 66]), they introduce another form of friction especially for capturing the hierarchical structure in development work [33]. Instead, continued research into improving the accuracy of automatically inferred tasks is necessary to support an intuitively proactive workspace [68, 116]. One limitation with existing approaches is they generally only have access to either high-level information across applications like the application title, or detailed information only within a single application. A new direction could be to incorporate the context captured by proactive workspaces as developers interact with different kinds of resources, using them as cues about when developers start and stop tasks. For example, an error message in the IDE could indicate the start of a bug fixing task within the developer's larger task of implementing a new feature.

6.4.3 Explainability

Although we believe proactive workspaces will relieve developers of unnecessary work, prior research has shown that trust is a key dimension for adaptive systems [119]. For example, current LLM-based approaches have challenges with correctness and explainability of their automatically generated content [104]. While having a tool flawlessly adapt to the developer as

they work is a lofty goal, new work will be required to help communicate these customizations to developers, and allow them to skip them, when the developer does not believe the customizations will be helpful. This is a challenging problem as this awareness must not be more expensive or intrusive than having the developer manually perform the work themselves.

Chapter 7

Conclusion

Developers rely on their tools to complete their tasks. While new and more capable tools continue to be introduced to support increasingly complex development tasks, the glue between these tools, the workspace, has largely remained unchanged. In our initial work examining developer's workflows, we found that developers decompose their high-level tasks into goals which they operationalize using low-level actions. However, the tool-centric design of current workspaces does not align with the cross-application workflows used by developers to complete their tasks which creates unnecessary work for developers. This unnecessary work acts a friction that impacts developers' progress on their core tasks. Specifically, we identified that developers encounter translation friction operationalizing their goals as low-level actions, integration friction incorporating information across dispersed resources, and access friction navigating across and within independent applications (RQ1).

Focusing on integration and access frictions, we devised two approaches that capture and share context across developer tools to help developers locate information more easily. Helm's centralized resource-centric approach addresses access friction by enabling developers to navigate directly between their resources. Helm uses information about the resources developers access and their active project as context to recommend relevant resources to the developer. During an exploratory study, Helm was able to recommend the vast majority of resources developers had previously opened, reducing the effort for them to re-find resources in the workspace. When resuming a task, Helm supported developers re-establish the required resources in the correct locations for their current project. Developers were able to interact with many of these recommendations without having to switch away from their current tool and resource, helping them to stay focused on their current task (RQ2).

Scout addresses integration friction that developers encounter when locating suitable APIs across the IDE and web browser by integrating web search into the IDE. Using information about the developer's code, Scout extracts and ranks API signatures contained within Stack Overflow search

result pages. By tailoring the search results using a compact API signature representation, Scout enables developers to directly locate and integrate API solutions into their code. By tailoring the search results using a compact API signature representation, Scout enables developers to directly locate and integrate API solutions into their code within the IDE. Through a controlled user study, we found Scout significantly reduced the amount of navigation required for developers to locate the API information necessary to complete their tasks (RQ3).

Current tool-centric designs force developers to manually overcome a variety of frictions induced by the diverse set of tools they use to complete their tasks. We believe that, by exposing the key identifiers, actions, and historical information developers have interacted with across their tools, workspaces could proactively align with developer's tasks helping them to be more productive. Based on the Helm and Scout approaches, and the findings from the studies, we identified proactive behaviours, including automatically setting working locations, supporting project-based tool histories, and linking resources across tools, that would help developers more easily coordinate and locate information across their existing tools.

7.1 Key Takeaways

We conclude this work with key takeaways meant to facilitate further work addressing friction in developers' workspaces.

7.1.1 The ways developers use their tools is unique but consistent.

Workflows are highly personal and are the result of each developer's unique experience, working style, and tool preferences. In particular, we observed that developers largely follow the same high-level steps to complete tasks, but their concrete actions varied significantly within these steps (e.g., when developers considered looking at source code versus documentation). The consistency of the high-level steps means that it is possible for relatively simple approaches to help address friction at a broad level by incorporating pre-conditions when helping coordinate tools (e.g., developers will read a bug report before fixing the code). Codifying (or learning) these general pre-conditions across developers and tasks is one way researchers could think about reducing friction in developer workspaces in a relatively top-down, tool-agnostic way. This approach is appealing to handle the combinatorial explosion of tools used across all developers. At the same time, developers'

unique workflows also point to more tailored approaches which take into account the specific actions developers perform across their comparatively small and stable set of tools. One way approaches could account for these unique workflows is by considering the *sequence* in which developers interact with their tools. This would enable subsequent tools to use knowledge about which tool the developer was using previously to help them identify the developer's current needs. For example, a developer's needs when opening a tool for the first time are often substantially different than when the developer subsequently returns to the tool, and information about the developer's tool usage sequence could help to make the resources and actions relevant to the developer's needs more immediately accessible. Supporting this kind of information sharing across the myriad of tools developers use in practice is not likely to be feasible using direct tool-to-tool communication. Identifying a general and robust communication framework within the workspace will be necessary to facilitate this communication.

7.1.2 Implications of proactive workspace recommendations on LLMs.

Accessing information in the workspace is costly for developers, both in terms of the mechanical effort required to navigate between resources, and the cognitive effort required to orient and process navigational cues in the workspace. Recommendations provide a mechanism to short-circuit this costly process by offering developers more direct access to information in the workspace. However, it can be challenging to identify ideal recommendations across developer's diverse workflows. Information about the actions developers perform across their tools can provide context to scope the recommendations. Even when this information is not fully sufficient to always ensure completely accurate recommendations, careful presentation of the recommendations enable developers to leverage their mental models to overcome incomplete or non-optimally ordered recommendations. For recommendations which do not contain the required resource, developers can still navigate closer to the resource by directly following a related resource recommendation, since most information is organized hierarchically in the workspace. For example, a code file recommendation could get the developer directly into the right project directory, while a bug report recommendation would get the developer into the right GitHub location to view related reports. The presentation of recommendations can also mitigate effects of low precision, where developers have to consider many recommendations, by making apparent relationships between resources that are not

otherwise represented in the workspace; for example, by bringing together different kinds of information or by surfacing different versions of the same kind of information. These relationships can help developers reason about and guide developers to the required resource in fewer navigational steps than manually navigating and inspecting each resource independently.

LLMs offer an alternative way for developers to access recommendations in their workspace. While LLMs are a significant step towards reducing friction, they are still individual tools that are bound by the same independent design as conventional tools. This means developers still need to coordinate LLM-based tools in their workflows. Providing additional contextual information from the developer's other tools could improve LLM recommendations. For example, an LLM-based commit message generator in the developer's IDE only has access to information about changed files. By sharing information from the developer's issue management tool, the LLM could also include information from the issue report such as the issue identifier.

Ultimately, this thesis has shown that it is possible to automatically gather workflow data from developers as they work and provide lightweight, proactive recommendations to reduce friction as they complete their tasks. These findings provide guidance for future researchers aiming to further support developers through more coordinated, but still independent tools, thereby improving developers' overall productivity.

Bibliography

[1] `https://github.com/features/copilot`. Accessed Jan 21, 2024.

[2] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. What do developers use the crowd for? a study using stack overflow. *IEEE Software*, 34(2):53–60, March 2017.

[3] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma. Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, 2020.

[4] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. CAPS: A supervised technique for classifying Stack Overflow posts concerning API issues. *Empirical Software Engineering (EMSE)*, 25(2):1493–1532, March 2020.

[5] T. M. Ahmed, W. Shang, and A. E. Hassan. An Empirical Study of the Copy and Paste Behavior during Development. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 99–110, 2015.

[6] Abdulaziz Alaboudi and Thomas D. LaToza. An Exploratory Study of Live-Streamed Programming. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 5–13, October 2019.

[7] Zakieh Alizadehsani, Enrique Goyenechea Gomez, Hadi Ghaemi, Sara Rodríguez González, Jaume Jordan, Alberto Fernández, and Belén Pérez-Lancho. Modern Integrated Development Environment (IDEs). In Juan M. Corchado and Saber Trabelsi, editors, *Sustainable Smart Cities and Territories*, pages 274–288, 2022.

[8] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *Proceedings of the Inter-*

national Conference on Software Analysis, Evolution and Reengineering (SANER), volume 1, pages 124–134, 2016.

[9] Saulius Astromskis, Gabriele Bavota, Andrea Janes, Barbara Russo, and Massimiliano Di Penta. Patterns of developers behaviour: A 1000-hour industrial study. 132:85–97, 2017.

[10] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4(3):114–123, May 2009.

[11] Liam Bannon, Allen Cypher, Steven Greenspan, and Melissa L. Monty. Evaluation and Analysis of Users' Activity Organization. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 54–57, 1983.

[12] Lingfeng Bao, Zhenchang Xing, Xinyu Wang, and Bo Zhou. Tracking and Analyzing Cross-Cutting Activities in Developers' Daily Work. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 277–282, 2015.

[13] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Ahmed E. Hassan. Inference of Development Activities from Interaction with Uninstrumented Applications. *Empirical Software Engineering (ESE)*, 23(3):1313–1351, 2018.

[14] Jakob E. Bardram, Steven Jeuris, Paolo Tell, Steven Houben, and Stephen Voida. Activity-centric Computing Systems. *Communications of the ACM*, 62(8):72–81, July 2019.

[15] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. Exempla gratis (E.G.): Code examples for free. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1353–1364, 2020.

[16] Patricia Bazeley. *Complementary Analysis of Varied Data Sources*, chapter 5, pages 91–125. 2022.

[17] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 125–134, 2010.

[18] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *ESECFSE*, pages 179–190, 2015.

[19] Michael S. Bernstein, Jeff Shrager, and Terry Winograd. Taskposé: Exploring Fluid Boundaries in an Associative Window Visualization. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 231–234, 2008.

[20] R. Bisiani, F. Lecouat, and V. Ambriola. A tool to coordinate tools. *IEEE Software*, 5(6):17–25, 1988.

[21] Caspar Boekhoudt. The Big Bang Theory of IDEs: Pondering the vastness of the ever-expanding universe of IDEs, you might wonder whether a usable IDE is too much to ask for. 1(7):74–82, 2003.

[22] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 455–464, 2010.

[23] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 2503–2512, 2010.

[24] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 513–522, 2010.

[25] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1589–1598, 2009.

[26] Virginia Braun and Victoria Clarke. *Thematic Analysis: A Practical Guide*. 2022.

[27] Fred P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[28] Katriina Byström and Preben Hansen. Conceptual framework for tasks in information studies. 56(10):1050–1061, 2005.

[29] Karsten Böhm and Lisa-Maria Schedlberger. The use of generative AI in the domain of human creations-a case for co-evolution? In *Proceedings of the International Conference on Socio-Technical Perspectives in IS (STPIS)*, 2023.

[30] Eduardo C. Campos, Lucas B. L. de Souza, and Marcelo de A. Maia. Searching crowd knowledge to recommend solutions for API usage tasks. *Journal of Software: Evolution and Process*, 28(10):863–892, July 2016.

[31] John M. Carroll and Caroline Carrithers. Training wheels in a user interface. 27(8):800–806, 1984.

[32] Joseph Chee Chang, Nathan Hahn, Yongsung Kim, Julina Coupland, Bradley Breneisen, Hannah S Kim, John Hwong, and Aniket Kittur. When the Tab Comes Due:Challenges in the Cost Structure of Browser Tab Usage. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1–15, 2021.

[33] S. Chattopadhyay, N. Nelson, Y. Ramirez Gonzalez, A. Amelia Leon, R. Pandita, and A. Sarma. Latent Patterns in Activities: A Field Study of How Developers Manage Context. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 373–383, 2019.

[34] Michael J. Coblenz, Amy J. Ko, and Brad A. Myers. JASPER: An Eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 65–69, 2006.

[35] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? Designing and Evaluating Screening Questions for Online Surveys with Programmers. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 537–548, 2021.

[36] Robert DeLine and Kael Rowan. Code canvas: Zooming towards better development environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 2, pages 207–210, 2010.

[37] Anton N. Dragunov, Thomas G. Dietterich, Kevin Johnsrude, Matthew McLaughlin, Lida Li, and Jonathan L. Herlocker. TaskTracer: A Desktop Environment to Support Multi-tasking Knowledge Workers. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, pages 75–82, 2005.

[38] Tore Dybå, Vigdis By Kampenes, and Dag I. K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 2006.

[39] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 175–184, 2010.

[40] G. Gao, F. Voichick, M. Ichinco, and C. Kelleher. Exploring Programmers API Learning Processes: Collecting Web Resources as External Memory. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, 2020.

[41] D. R. Garrison, M. Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, January 2006.

[42] Marko Gasparic, Gail C. Murphy, and Francesco Ricci. A Context Model for IDE-based Recommendation Systems. 128(C):200–219, 2017.

[43] Max Goldman and Robert C. Miller. Codetrail: Connecting source code and web resources. 20(4):223–235, 2009.

[44] Victor M. González, Gloria Mark, and Gloria Mark. "Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 113–120, 2004.

[45] Chase Greco, Tyler Haden, and Kostadin Damevski. StackInTheFlow: Behavior-Driven Recommendation System for Stack Overflow Posts.

In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 5–8, May 2018.

[46] Thomas RG Green. Cognitive dimensions of notations. pages 443–460, 1989.

[47] Karl Gyllstrom and Craig Soules. Seeing is retrieving: Building information context from what the user sees. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, pages 189–198, 2008.

[48] David Harel, Guy Katz, Rami Marelly, and Assaf Marron. Wise computing: Toward endowing system development with proactive wisdom. *Computer*, 51(2):14–26, 2018.

[49] Björn Hartmann, Mark Dhillon, and Matthew K. Chan. HyperSource: Bridging the Gap Between Source and Code-related Web Sites. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 2207–2210, 2011.

[50] Melanie Hartmann, Daniel Schreiber, and Max Mühlhäuser. AUGUR: Providing Context-aware Interaction Support. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 123–132, 2009.

[51] Austin Z. Henley and Scott D. Fleming. The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 2511–2520, 2014.

[52] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: Finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 13–22, 2007.

[53] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[54] Andre Hora. Googling for Software Development: What Developers Search For and What They Find. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 317–328, 2021.

[55] Donghan Hu and Sang Won Lee. ScreenTrack: Using a Visual History of a Computer Screen to Retrieve Documents and Web Pages. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1–13, 2020.

[56] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 293–304, 2018.

[57] Karen E. Huff and Victor R. Lesser. A plan-based intelligent assistant that supports the software development. *ACM SIGSOFT Software Engineering Notes*, 13(5):97–106, 1988.

[58] Giulio Jacucci, Pedram Daee, Tung Vuong, Salvatore Andolina, Khalil Klouche, Mats Sjöberg, Tuukka Ruotsalo, and Samuel Kaski. Entity Recommendation for Everyday Digital Tasks. 28(5):29:1–29:41, 2021.

[59] Steven Jeuris, Steven Houben, and Jakob Bardram. Laevo: A temporal desktop interface for integrated knowledge work. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 679–688, 2014.

[60] Xiaoyu Jin and Nan Niu. Short-Term Revisit during Programming Tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 322–324, 2017.

[61] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. Bespoke tools: Adapted to the concepts developers know. In *ESECFSE*, pages 878–881, 2015.

[62] Samia Kabir, David N. Udo-Imeh, Bonan Kou, and Tianyi Zhang. Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions, 2023.

[63] Victor Kaptelinin. UMEA: Translating Interaction Histories into Project Contexts. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 353–360, 2003.

[64] Victor Kaptelinin and Richard Boardman. Toward Integrated Work Environments: Application-Centric versus Workspace-Level Design. In Victor Kaptelinin and Mary Czerwinski, editors, *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments*. 2007.

[65] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. The IDE portability problem and its solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 152–162, 2016.

[66] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 159–168, 2005.

[67] Katja Kevic and Thomas Fritz. Automatic search term identification for change tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 468–471, 2014.

[68] Katja Kevic and Thomas Fritz. Towards activity-aware tool support for change tasks. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 171–182, 2017.

[69] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy - a code-to-code search engine. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 946–957, 2018.

[70] Amy J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 344–353, 2007.

[71] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 151–158, 2004.

[72] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *Transactions on Software Engineering (TSE)*, 32(12):971–987, 2006.

[73] Thomas D. LaToza and Brad A. Myers. Designing Useful Tools for Developers. In *Proceedings of the ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 45–50, 2011.

[74] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 492–501, 2006.

[75] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. Scents in Programs:Does Information Foraging Theory Apply to Program Maintenance? In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 15–22, 2007.

[76] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. 39(2):197–215, 2013.

[77] Clayton Lewis and Cathleen Wharton. Chapter 30 - Cognitive Walkthroughs. In Marting G. Helander, Thomas K. Landauer, and Prasad V. Prabhu, editors, *Handbook of Human-Computer Interaction*, pages 717–732. Second edition, 1997.

[78] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. What help do developers seek, when and how? In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 142–151, 2013.

[79] Jiakun Liu, Sebastian Baltes, Christoph Treude, David Lo, Yun Zhang, and Xin Xia. Characterizing search activities on stack overflow. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 919–931, August 2021.

[80] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. API-Related Developer Information Needs in Stack Overflow. *Transactions on Software Engineering (TSE)*, 48(11):4485–4500, November 2021.

[81] Walid Maalej. Task-First or Context-First? Tool Integration Revisited. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 344–355, 2009.

[82] Walid Maalej and Alexander Sahm. Assisting Engineers in Switching Artifacts by Using Task Semantic and Interaction History. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 59–63, 2010.

[83] Sonal Mahajan, Negarsadat Abolhassani, and Mukul R. Prasad. Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1052–1064, November 2020.

[84] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005.

[85] Gloria Mark, Victor M. Gonzalez, and Justin Harris. No task left behind? examining the nature of fragmented work. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 321–330, 2005.

[86] Gloria Mark, Shamsi T. Iqbal, Mary Czerwinski, Paul Johns, and Akane Sano. Neurotics Can't Focus: An in situ Study of Online Multitasking in the Workplace. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1739–1744, 2016.

[87] Arthur Marques, Nick C. Bradley, and Gail C. Murphy. Characterizing Task-Relevant Information in Natural Language Software Artifacts. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–487, 2020.

[88] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 111–120, 2011.

[89] André N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. The Work Life of Developers: Activities, Switches and Perceived Productivity. *Transactions on Software Engineering (TSE)*, 43(12):1178–1193, 2017.

[90] André N. Meyer, Thomas Zimmermann, and Thomas Fritz. Characterizing Software Developers by Perceptions of Productivity. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 105–110, 2017.

[91] R. Minelli, A. Mocci, and M. Lanza. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 25–35, 2015.

[92] Roberto Minelli, Andrea Mocci, and Michele Lanza. The Plague Doctor: A Promising Cure for the Window Plague. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 182–185, 2015.

[93] Jenny Morales, Federico Botella, Cristian Rusu, and Daniela Quiñones. How "Friendly" Integrated Development Environments Are? In *Social Computing and Social Media. Design, Human Behaviour, and Analystics (SCSM)*, pages 80–91, 2019.

[94] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[95] Gail C Murphy. Beyond Integrated Development Environments: Adding Context to Software Development. In *ICSE-NIER*, pages 73–76, 2019.

[96] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubranić. The emergent structure of development tasks. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'05, pages 33–48, 2005.

[97] Fiona Fui-Hoon Nah. A study on tolerable waiting time: How long are Web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, May 2004.

[98] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. In-IDE Generation-based Information Support with a Large Language Model, 2023.

[99] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 25–34, September 2012.

[100] Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2022.

[101] Nuria Oliver, Mary Czerwinski, Greg Smith, and Kristof Roomp. RelAltTab: Assisting Users in Switching Windows. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, pages 385–388, 2008.

[102] Nuria Oliver, Greg Smith, Chintan Thakkar, and Arun C. Surendran. SWISH: Semantic Analysis of Window Titles and Switching History. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, pages 194–201, 2006.

[103] Stephen Oney and Joel Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 2697–2706, 2012.

[104] Ipek Ozkaya. The next frontier in software development: AI-augmented software development processes. *IEEE Software*, 40(4):4–9, 2023.

[105] Cole S. Peterson, Jonathan A. Saddler, Natalie M. Halavick, and Bonita Sharif. A Gaze-Based Exploratory Study on the Information Seeking Behavior of Developers on Stack Overflow. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–6, May 2019.

[106] Jan Pilzer, Raphael Rosenast, André N. Meyer, Elaine M. Huang, and Thomas Fritz. Supporting Software Developers' Focused Work on Window-Based Desktops. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1–13, 2020.

[107] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? How production bias affects developers' information foraging during debugging. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 11–20, 2015.

[108] David Piorkowski, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 97–108, 2016.

[109] David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 3063–3072, 2013.

[110] Peter Pirolli and Stuart Card. Information foraging in information access environments. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 51–58, 1995.

[111] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 102–111, 2014.

[112] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 94–105, 2017.

[113] Sruti Srinivasa Ragavan, Mihai Codoban, David Piorkowski, Danny Dig, and Margaret Burnett. Version Control Systems: An Information Foraging Perspective. *Transactions on Software Engineering (TSE)*, 47(8):1644–1655, 2021.

[114] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 465–475, 2018.

[115] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. Automatic query reformulation for code search using crowdsourced knowledge. *Empirical Software Engineering (EMSE)*, 24(4):1869–1924, August 2019.

[116] Tye Rattenbury and John Canny. CAAD: An Automatic Task Support System. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pages 687–696, 2007.

[117] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison-Wesley, 1990.

[118] George Robertson, Eric Horvitz, Mary Czerwinski, Patrick Baudisch, Dugald Ralph Hutchings, Brian Meyers, Daniel Robbins, and Greg Smith. Scalable Fabric: Flexible Task Management. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*, pages 85–89, 2004.

[119] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer, 2014.

[120] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: An exploratory study. 30(12):889–903, 2004.

[121] Tobias Roehm and Walid Maalej. Automatically Detecting Developer Activities and Problems in Software Development Work. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1261–1264, 2012.

[122] David Roethlisberger, Oscar Nierstrasz, and Stephane Ducasse. Autumn Leaves: Curing the Window Plague in IDEs. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 237–246, 2009.

[123] G. Di Rosa, A. Mocci, and M. D'Ambros. Visualizing Interaction Data Inside Outside the IDE to Characterize Developer Productivity. In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 38–48, September 2020.

[124] Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. PostFinder: Mining Stack Overflow posts to support software developers. *Information and Software Technology*, 127, November 2020.

[125] Daniel Russo. Recruiting Software Engineers on Prolific. In *International Workshop on Recruiting Participants for Empirical Software Engineering*, 2022.

[126] Roy Rutishauser, André A. Meyer, Reid Holmes, and Thomas Fritz. Semi-Automatic, Inline and Collaborative Web Page Code Curations.

In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1866–1877, 2023.

[127] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: A case study. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 191–201, 2015.

[128] Nicholas Sawadsky and Gail C. Murphy. Fishtail: From task context to source code examples. In *Proceedings of the Workshop on Developing Tools as Plug-ins*, pages 48–51, 2011.

[129] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. Reverb: Recommending Code-related Web Pages. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 812–821, 2013.

[130] Adam C. Short and Austin Z. Henley. Towards an Empirically-Based IDE: An Analysis of Code Size and Screen Space. In *VLHCC*, pages 199–203, 2019.

[131] Howard Shrobe, Boris Katz, and Randall Davis. Towards a Programmer's Apprentice (again). In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 4062–4066, 2015.

[132] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *Transactions on Software Engineering (TSE)*, 34(4):434–451, 2008.

[133] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 21–36, 1997.

[134] David C. Smith, Charles Irby, Ralph Kimball, and Eric Harslem. The Star User Interface: An Overview. In Per Brinch Hansen, editor, *Classic Operating Systems: From Batch Processing To Distributed Systems*, pages 460–490. 2001.

[135] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[136] Diomidis Spinellis and Stephanos Androutsellis-Theotokis. Software Development Tooling: Information, Opinion, Guidelines, and Tools. 31(6):21–23, 2014.

[137] Aaron Springer and Steve Whittaker. Progressive disclosure: Empirically motivated approaches to designing effective transparency. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 107–120, 2019.

[138] J. Stylos and B.A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 195–202, 2006.

[139] Jeffrey Stylos, Brad A. Myers, and Andrew Faulring. Citrine: Providing intelligent copy-and-paste. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 185–188, 2004.

[140] Craig Tashman. WindowScape: A Task Oriented Window Manager. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 77–80, 2006.

[141] E. Thiselton and C. Treude. Enhancing Python Compiler Error Messages via Stack. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, September 2019.

[142] Robin R. Vallacher and Daniel M. Wegner. Action identification theory. In *Handbook of Theories of Social Psychology, Vol. 1*, pages 327–348. 2012.

[143] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. The sky is not the limit: Multitasking across GitHub projects. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 994–1005, 2016.

[144] Gina Venolia. Bridges between silos: A microsoft research project. January 2005.

[145] Petcharat Viriyakattiyaporn and Gail C. Murphy. Improving program navigation with an active help system. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 27–41, 2010.

[146] Manuela Waldner, Stefan Bruckner, and Ivan Viola. Graphical Histories of Information Foraging. In *Proceedings of the Nordic Conference on Human-Computer Interaction (NordiCHI)*, pages 295–304, 2014.

[147] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

[148] Yingxu Wang and Vincent Chiew. On the cognitive process of human problem solving. *Cognitive Systems Research*, 11(1):81–92, 2010.

[149] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017.

[150] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. AnswerBot: Automated generation of answer summary to developers' technical questions. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 706–716, 2017.

[151] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 460–471, 2014.

[152] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example Overflow: Using social media for code recommendation. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 38–42, June 2012.

[153] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. Bing developer assistant: Improving developer productivity by recommending sample code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 956–961, 2016.

[154] Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. Chatbot4QR: Interactive Query Refinement for Technical Question Retrieval. *Transactions on Software Engineering (TSE)*, 48(4):1185–1211, April 2022.

[155] Shengdong Zhao, Fanny Chevalier, Wei Tsang Ooi, Chee Yuan Lee, and Arpit Agarwal. AutoComPaste: Auto-completing text as an alternative to copy-paste. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI)*, pages 365–372, 2012.

[156] Lijie Zou and Michael W. Godfrey. An industrial case study of Coman's automated task detection algorithm: What Worked, What Didn't, and Why. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 6–14, 2012.

[157] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418, 2003.

Appendix A

Supporting Material for Studies

The study instruments, tools, data, and analysis scripts that support Chapters 3, 4, 5 of this thesis are organized into corresponding modules which are available at `https://doi.org/10.17605/OSF.IO/H42PU`.

A.1 Developer Workflow Transcripts

The anonymous transcribed data is available online for further analysis.

A.2 Helm Replication Package

A tool demo and details regarding the interview are available online in our replication package.

A.3 Scout Replication Package

Our study instruments, data, analysis scripts, and demo are available online in our replication package.